SystemC Kernel Extensions for Heterogeneous System Modeling

A Framework for Multi-MoC Modeling & Simulation

By
Hiren D. Patel and Sandeep K. Shukla

Kluwer Academic Publishers

SYSTEMC KERNEL EXTENSIONS FOR HETEROGENEOUS SYSTEM MODELING

SystemC Kernel Extensions for Heterogeneous System Modeling

A Framework for Multi-MoC Modeling & Simulation

by

Hiren D. Patel
Virginia Polytechnic and State University,
Blacksburg, VA, U.S.A.

and

Sandeep K. Shukla
Virginia Polytechnic and State University,
Blacksburg, VA, U.S.A.

KLUWER ACADEMIC PUBLISHERS
NEW YORK, BOSTON, DORDRECHT, LONDON, MOSCOW

eBook ISBN: 1-4020-8088-3
Print ISBN: 1-4020-8087-5

Created in the United States of America

Visit Springer's eBookstore at: http://ebooks.kluweronline.com
and the Springer Global Website Online at: http://www.springeronline.com

To my parents

Maya D. Patel
and
Dhanji K. Patel

Hiren D. Patel

To my friends

Tushar Saxena,
Arush Saxena
and their families

Sandeep K. Shukla

Contents

List of Figures

List of Tables

Foreword

"Entia non sunt multiplicanda praeter necessitatem"
(No more things should be presumed to exist than are absolutely necessary) William Occam, ***Quodlibeta***, c. 1324, V, Q.i

"There never were in the world two opinions alike, no more than two hairs or two grains; the most universal quality is diversity"
Montaigne, 1533-1592

We have seen a rapid growth of interest in the use of SystemC for system level modeling since the language first emerged in the late 1990's. As the language has evolved through several generations of capability, designers and researchers in a wide variety of systems and semiconductor design houses, and in many academic institutions, have built models of systems or partial systems and explored the capabilities and limitations of SystemC as a base on which to build system-level models. Many of these models have been used as part of the specification and design of systems which have subsequently been put into production; in these cases, we can justifiably conclude that SystemC-based models have seen "tape-outs" even though the language has been used for answering early, high-level, system modeling and configuration questions, rather than as the language of detailed implementation.

Another key role that has been played by the SystemC community has been to experiment with many concepts of system level modeling using SystemC as a base. These experiments have included adding mixed-signal, continuous-time model solving to the discrete time concepts of SystemC; developing various notions for transaction-level modeling at intermediate abstraction levels between untimed functional models and time, signal and pin-accurate RTL models; connecting SystemC to network simulators; developing synthesis capabilities; and many other ex-

periments. Some of these have contributed ideas directly to the evolution of SystemC which is being carried out by working groups under the auspices of the Open SystemC Initiative (OSCI); others have been research projects whose results and code have been freely shared with others in the SystemC community.

It has been important to recognize that the official development of SystemC within OSCI and in the future as part of the IEEE (when the language donation process from OSCI to IEEE has been concluded, sometime in 2004), is but one stream of language development and evolution. One of the key virtues of using an open source or community source model with SystemC has been in making the source code of the reference class libraries and simulator kernel available to the community for experimentation. This allows academic researchers, system designers, and the commercial EDA tools industry all great latitude in seeking ways to optimize SystemC for particular purposes, use models and design domains. Whether or not the resulting work is put back into SystemC, it all serves to grow the community and knowledge base of effective methods for carrying out system level design and using SystemC. Some improvements may remain the domain of proprietary in-house or commercial EDA tools; some may remain applicable to narrow groups of researchers and developers in a very particular design niche. And some may influence the future evolution of the language. Since the beginning, the reference SystemC implementation has been intended to be just that- a reference, for use, experiment and to provoke thought; not to be "the last word" in efficient implementation (although it has exceeded all reasonable expectations for quality, efficiency and usefulness).

In this sense, SystemC has thrown down a challenge to the community at large to seek methods to optimize it for particular uses and for system-level modeling in the widest sense. It is thus extremely gratifying to see that Hiren Patel and Sandeep Shukla, of the FERMAT Research Lab, Center for Embedded Systems for Critical Applications, Virginia Polytechnic Institute and State University, have risen to that challenge.

This new book, *SystemC Kernel Extensions for Heterogeneous System Modeling*, is based on the very recent research work by Mr. Patel and Professor Shukla into improving the efficiency and heterogeneity of SystemC for system level design involving multiple models of computation (MoC). It marks a major step forward in looking at extensions to the basic SystemC modeling constructs and kernel to handle more complex multi-domain modeling issues in an efficient manner. Although it is quite possible, as discussed by the authors, to build multiple domain system models using the basic discrete event model of computation supported by the reference SystemC implementation from OSCI, this is far from

efficient for systems such as synchronous data flow that can be statically scheduled. Finding, implementing, and sharing improved methods and algorithms within the context of the SystemC community is an excellent way of meeting the challenge laid down by the reference simulator. This monograph is a concise summary of how that challenge has been met by the authors. It will thus be of interest to anyone wanting to extend SystemC for more efficient multi-domain simulation, or commercial or internal tools groups wanting to build domain-specific SystemC based simulators that have improved performance.

In their introduction, the authors give an overview of system-level design, system level modeling, models of computation, and their approach to heterogeneous modeling and the major contributions of their research. This introduction is a useful and quick summary of the thrust of the research, and serves to motivate the goals of the work as a whole.

This is followed by two key chapters that flesh out the details behind system level modeling approaches and give an analysis of the SystemC discrete-event kernel. They review both multi-domain system modeling examples such as Ptolemy II, to illustrate the notions of multiple models of computation, and discuss the details of the SystemC modeling and simulation approach.

The next several chapters are the heart of the research, covering three different domain models and their implementation by strategic modifications to the SystemC discrete event kernel to allow multiple MoC domains. These include synchronous data flow, Hoare's communicating sequential processes (CSPs), and finite state machines. Each of these modified kernels is described using extensive examples of implementation code, discussed in detail in each chapter. Where there is insufficient room to publish the complete code, and for the convenience of the readers and experimenters, reference is given to the FERMAT lab web site, where the full source listings have been made available.

The remaining chapters cover diverse topics: an API to the modified simulation kernel; several heterogeneous system model examples, building towards one which involves CSP, FSM, SDF and DE kernels, and measured efficiency results which indicate a baseline of speedup possible through using such methods which provides a floor for what is achievable in optimizing domain-specific SystemC. It is notable that those MoCs that are amenable to static scheduling, such as SDF, show by far the greatest speedup, as one would expect.

Finally, two appendices provide a discussion of the QuickThreads coroutine execution model in the SystemC kernel, and notes on configuring and building the multiple kernel models. A useful set of references to

various standard sources on languages, SystemC modeling, and models of computation, provides ample pointers to further reading.

In summary, this monograph is a useful step forward in devising, proving and documenting SystemC simulation kernel enhancements for multi-domain system modeling. We hope that this response to the challenges posed by SystemC will be answered by other researchers who can take this kind of work forward to even greater efficiencies, and that the contributions which the authors plan to make to the SystemC language committees will have a positive impact on the future evolution of SystemC.

Grant Edmund Martin, Chief Scientist
Tensilica, Inc.
Santa Clara, California, April 2004

Preface

We believe that one important step towards mitigating the current *productivity crisis* in the semi-conductor industry can be achieved by raising the level of abstraction at which designers encode their conceptual designs into models. The current practice of using the structured RTL as the language of design entry will not allow us to cope with the rising complexity of designs. In fact, it is quite clear for some time, as stated by many industry experts explicitly in many forums, that time for higher abstraction level above RTL has already come. A decade or so ago, we have moved abstraction level from schematics to RTL, and in the meantime, continual improvement in technology faithful to Moore's law, has increased the design complexity orders of magnitude higher. It is time for another move. However, to achieve this, we need languages and frameworks which help us make our design entry at higher abstraction level, and in the appropriate models of computation. From the year 1999 onwards, SystemC and other C++ based languages emerged with the promise of higher abstraction level, but did not hold onto their promise to our disappointment. The SystemC reference implementations for SystemC versions 0.9 and 1.0 were completely based on discrete-event simulation semantics *a la* VHDL. SystemC 2.0 introduced the notion of events, channels, and interfaces which looked like an improvement over 1.0, however, it really did not provide the infrastructure for *heterogeneity* that we felt was needed for truly heterogeneous system modeling. At the same time, Ptolemy II [25] inspired our imagination and desire to have a SystemC based framework similar to Ptolemy II, but fundamentally distinct because the scope and target of our desirable system is different from that of Ptolemy II, which is mainly concerned with embedded software system synthesis from actor-oriented modeling. Moreover, the industry traction of SystemC and the hopes placed in SystemC as a language of choice for system-level-modeling can only be sustained by

adding capabilities to SystemC to make it useful, rather than replacing it with Ptolemy II or other frameworks. Of course, a lot of concepts can be borrowed from Ptolemy II to achieve this goal.

This book is a result of an almost two year endeavor on our part to understand how SystemC can be made useful for system level modeling at higher levels of abstraction. Making it a truly heterogeneous modeling language and platform, for hardware/software co-design as well as complex embedded hardware designs has been our focus in the work reported in this book. Towards this aim, we first implemented a separate kernel adjoined to the simulation kernel of the SystemC reference implementation, so that Synchronous Data Flow (SDF) [25] models can be modeled and simulated with the new kernel. This kernel can coexist with the reference kernel since the rest of the components of the system not conforming to SDF semantics continue to be modeled and simulated by the reference kernel. The reference kernel is based on the Discrete-Event (DE) simulation semantics, making it hard to model and efficiently simulate SDF. This implementation and subsequent experiments showed significant improvements in simulation speed for models which are most naturally modeled in SDF style. This is because the computation in SDF models are statically schedulable, as opposed to dynamic scheduling needed on a DE kernel. Encouraged by these results, we undertook the project of extending this multi-kernel SystemC prototype towards more heterogeneity, resulting in prototype kernels for *Communicating Sequential Processes* (CSP) [28], and *Finite State Machine* (FSM) models of computation. The choice of extending in these two domains is obvious from the perspective of embedded systems modeling, since these two are the most commonly occurring models of computation in a hardware-software system rich in concurrency, and control path dominated components. The challenging part has been the integration of models which are simulated by distinct kernels.

Before we proceed further introducing the details of this book, we must make a disclaimer. Our current implementation which will be made available on our website [36] is certainly not industry strength, and we do not have the resources to make it so. All the programming done for this project is implemented by a single person, who is also tasked with writing thesis, papers, present this work at conferences, and take doctoral level classes. However, the reason for publication of this book is not to claim a full-fledged extension of SystemC with full heterogeneous capabilities. The reasons for spending an enormous number of hours on writing this book, despite lack of resources, and lack of industrial funding are summarized below.

At this point in time, we think that we have charted a pathway for extending SystemC towards true heterogeneity. We have successfully demonstrated how to add heterogeneity for directly designing distinct computational models appropriate for different components of a system, and how they can co-exist and co-simulate. We decided to share the status of this experiment and our findings with the rest of the EDA community with the hope that it would convince some readers to take up this momentum towards heterogeneous and hierarchical modeling framework based on SystemC, and develop it into a robust industry strength tool. This, we believe will make a difference in whether SystemC is adopted as a language of choice for System Level Modeling in the near future.

We expect this book to serve more as a convincing argument to persuade other researchers in the field to look into this paradigm, rather than as a prescription for solving the problem of system level modeling in its entirety.

Having made our disclaimer as to the robustness of our software, we nevertheless would urge the readers to download and experiment with our alternative kernels. We expect the readers to find deficiencies as well as to create wishlists of characteristics that one would like to have in a heterogeneous modeling framework. Such experiments are absolutely necessary steps towards a goal of eventually having a robust environment and language for system level modeling with heterogeneity and hierarchy as prime features.

In the rest of this preface we would like to answer some questions on the terminology that we often find confuse the students and engineers, due to varying usage in a variety of distinct contexts. The rest of the sections in this preface provide the readers with basic understanding of terms such as Models of Computation, heterogeneous and hierarchical modeling, and also discuss the desirable characteristics of modeling frameworks, namely *fidelity, expressiveness* and *robustness*.

What are Models of Computation?

The term *Model of Computation* (MoC) is a frequently used term in the Computer Science literature, in enough diverse contexts so as to confuse the readers. As a result, it is important that we clarify the context in which we use the term throughout this book, and also draw the reader's attention to the other alternative usages of this term. Of course, contextual variety does not mean the term is overloaded with distinct interpretations. The invariant in all of its usage archaic or modern, is that it refers to a formalized or idealized notion of computing rules.

Going back centuries, Euclid's algorithm for finding the greatest common divisor of two numbers can be idealized to a pencil and paper model

of computation. There, the only steps allowed were multiplication followed by subtraction, followed by comparison and repetitions of this sequence of steps until a termination condition is reached.

In the early history of computing, Charles Babbage [33] used mechanical gears to compute arithmetic, and hence his mechanical computer can be idealized by a model of arithmetic operations, where each primitive step of arithmetic is mapped to a manipulative action of mechanical gears and such. Early electronic computing machines, such as ENIAC [70], had a model of computation, where each action by the computer needed to be mapped to a physical wiring, which were models of programs in such early computers.

However, with the progress of computing and computer engineering, the complexity of computing systems has risen by orders of magnitude. At the same time, our ability to abstract out the essence of a computational system, leaving out the inessential coincidental mechanistic or other details has improved. We are able to formalize, often axiomatize the actions allowable by a computing process, and call it a model of computation.

Models of computation that allowed mathematicians to answer difficult questions posed by David Hilbert at the dawn of the 20th century, include Church's Lambda-Calculus [9], Post machine [67], and Turing Machine [66]. These provided universal models of computation that could express any computational step possible in the realm of computation, albeit, not always in the most efficient manner. With the advent of Scott-Rabin [58] automata theory, several restricted models of computation, such as finite state automata emerged. Understanding of their inexpressiveness and its gradual enhancement by adding more features to these models become apparent in the Chomsky hierarchy [8].

The von Neumann model of computation which promoted the idea of stored program computers (and is equivalent to Universal Turing Machine model) is now the most commonly used model for sequential computation. With the advances in computing, as we started exploiting overlapping sequential computations in the form of concurrency, and parallel computation in the form of parallelism, newer models of computations were necessitated. Communicating Sequential Processes (CSP) [28] invented by Tony Hoare, or Calculus of Concurrent Systems (CCS) [43] invented by Robin Milner, and other process algebraic models of computation paved the way for expressing concurrent and/or parallel computational models.

Computer scientists often distinguish between a denotational model and operational model of computation. A denotational model expresses a functional view over all possible states of the computation, whereas

an operational model spells out the steps of computation in an idealized model. However, which one of these two is more appropriate to idealize a computing system or a process depends on the intended applications of such an exercise.

Depending on the reason for abstracting a computing process into a model of computation, one can come up with different types of models. Researchers with focus on complexity theory, invented various models of Turing machine with resource bounds [29, 50] (e.g., Polynomial Time Deterministic Turing Machine vs. Polynomial Space bounded Turing Machine). When the major focus of the abstraction is to understand the parallel complexity, PRAM [31], and other idealized parallel machine models are proposed. The language theorists concentrate more on semantic aspects of computing and computational expressibility than on complexity issues. These gave rise to other classes of models of computation which include typed lambda calculus, process calculi, higher order calculi such as higher order lambda calculus [57] for sequential programming models, pi-calculus for concurrent programming models [59] etc.

Interestingly, for sequential models of computation, often expressiveness can be tied to complexity of computation, as can be found in the *descriptive complexity* theory. It can be shown that if a computational task is expressible by a certain formal logic, then the time or space complexity of that task can be precisely figured out. For example, all problems that can be stated in the existentially quantified second-order logic are in the complexity class NP [50]. However, in this book we do not delve into such complexity vs. expressiveness issues.

Other pragmatic models of computation were created by alternative ideas of computation, such as neural networks, DNA computing, molecular computing, quantum computing etc. These are also not our concern as far as the context and topic of this book is concerned.

Given this history of the term Model of Computation or MoC, the realm of discussion on these models remained confined in the domain of theoretical computer science until system designers of embedded systems, of system-on-chip, or of highly distributed heterogeneous systems started looking at the problem of abstraction, expressibility, and ease of modeling and simulation of highly diverse set of computational elements within a single system.

The most popular classification and comparison of models of computation in embedded system design context arose in the context of the Ptolemy and Ptolemy II projects [25] at the University of California at Berkeley. In a seminar paper by Lee and Vincentelli, a framework [39] of tagged signal models for comparison of various models of computa-

tion was proposed. Ptolemy project has been built around this notion of a variety of models of computation, which include various sequential models of computation such as FSM, Discrete-Time, Continuous Time models, as well as models of interaction and communicating entities, such as CSP, interaction models etc. Ptolemy II undoubtedly popularized the idea of viewing a complex embedded system as a collection of models which may belong to distinct models of computation (called domains in Ptolemy), and the idea of creating a framework in which efficient co-simulation of models from distinct domains is possible.

Another important work in the usage of Models of Computation in abstracting functionalities of complex heterogeneous system was done in the context of ForSyDe [15, 32] project in Sweden by Axel Jantsch and his group. We distinguished this work from the Ptolemy group's work as a difference between a denotational view vs. operational view of models of computation. Of course, such distinction have not been made earlier, but to avoid confusion among the readers as to the use of these two different classification of models of computation, we provide a structure to the different systems of classification.

A denotational view looks at a model of computation as a set of process constructors, and process composition operators, and provides denotational semantics to the constructors and operators. In [32] Axel Jantsch builds a classification based on the information abstraction perspective. The main models of computation in this view are

1 **Untimed Model of Computation:** where the timing information is abstracted so that no knowledge of time taken for computation or communication between computing processes is assumed. Since this MoC makes no assumption on time, the process constructors in this MoC can be used to construct processes which contain information on computation carried out and data to be transferred, without any regard to how much time it takes to carry them out.

2 **Synchronous Model of Computation:** where the timing information is abstracted to cycles of computation. This model has two distinct subdomains. The first one assumes perfect synchrony, and thereby assumes that computation and communication happens in zero time, and hence the only way time passes is through evaluation cycles. The computation in perfect synchrony assumption is expressed in terms of invariants between current value of a variable and its value in the next evaluation cycle. So a computation in this model is always a fixed point evaluation for enforcing the invariants. This model is the exact model used by Synchronous Programming paradigm used in Esterel, SIGNAL, Lustre and other synchronous

languages. The other subdomain is that of clocked synchronous model of computation, where the presence of a global clock enforces the boundaries of evaluation cycles, and therefore, the computation must take place within clock boundaries. This model of computation is suitable for digital hardware.

3 **Timed Model of Computation:** where timing information is not abstracted away. Exact timings of all computation and communications are modeled. This is the least abstract model of computation, and hence allows us to model a computing process to its minute details in terms of time taken to carry out any step.

This denotational classification abstracts out (i) all operational details as to how a computation step is carried out, (ii) how communication takes place between processes within the models (communication is abstracted to function calls and function applications), and (iii) timing information to various degrees.

The operational view of a classification however, classifies models of computation on how the computation takes place in terms of operational idiosyncrasies, as well as the way the communication takes place between processes within the same model of computation. The Ptolemy approach to classification of MoCs is exactly operational in that sense. From our perspective, this operational classification is quite orthogonal to the denotational classification.

Let us consider some examples of various MoCs or *domains* in Ptolemy II, which should clarify the distinction and orthogonality mentioned above.

Some of these domains are:

- **Continuous-Time:** In this MoC, the computation process performs analytical computation on continuous domains of real-numbers, through solving differential or integral equations or other analytical function computation. The communication between components performing other continuous-time computation is usually through variable sharing or through function calls.

- **Finite State Machine:** In this MoC, the computation is encapsulated in states, and state changes allow a different mode of computation to get activated. This is useful in describing control systems, or finite state control for hybrid computations.

- **Communicating Sequential Processes:** In this MoC, the distinction with respect to other domains arise not from the way computation is carried out, but rather from the communication mechanism

between processes within the same domain. The processes within the domain communicate via *rendez-vous* protocol, and the communication is synchronous.

- **Discrete Event Simulation:** In this domain, the computation is carried out in cycles, and each cycle consists of fixpoint computation for a selected subset of variables, which constitute the so called 'combinational' part of the system and this fixpointing is done through cycles known as 'delta cycles'. The communication in this domain is not specifically distinguished.

As one can see from the listing of these MoCs and their brief descriptions, these are classified according to their computational and communication characteristics from an operational perspective. One can also see the orthogonality, for example, by observing that the Communicating Sequential Processes can be untimed or timed, or even clock synchronous, as long as the rendez-vous protocol is maintained in the inter-process communication. Similarly the Finite State Machine can be timed, untimed, or clocked synchronous.

Frameworks for Expressing Models of Computation

In the light of our discussion on classification of models of computations (MoCs), we consider various frameworks that allow designers to build models of working systems (such as an embedded software/hardware system, or a system-on-chip). Such frameworks should facilitate modeling various components of such systems in MoC domains that are most appropriate for the level of details and the operational characteristics of the components. For example, if a system-on-chip design for a digital camera consists of a DSP processor, a microcontroller, and Analog-Digital and Digital-Analog converters, and some glue logics, each of these components may be suitable for modeling in different MoC domains. For example, the DSP component may be best modeled with a Synchronous Data Flow [52, 56, 54, 55] model, while the microcontroller may be modeled with precise clock accurate discrete-event computation models. We call a framework that allows such multi-MoC modeling and simulation, a **multi-MoC framework**.

Needless to say Ptolemy II [25] is such a multi-MoC framework, and so is ForSyDe [15]. Although each of these frameworks have their own degree of *fidelity* in such modeling. The *fidelity* of a multi-MoC framework can be defined as the degree of accuracy to which a theoretical model of computation can be modeled in the framework.

The multi-MoC framework in Ptolemy II requires the user to build the components in Java, using some characteristics of the target MoC

domain, and then placing the components in the appropriate domain under the control of a domain director, which schedules and guides the simulation of these components, and helps the components communicate according to domain specific communication rules. In ForSyDe, the components are built using functional programming language Haskell, suitable for the denotational mode of expressing the computational structure of the components. Recently, the same denotational MoC framework has been implemented [1] in another functional programming language SML. The facilities of multi-MoC modeling provided by these frameworks are designed in a way, so that the multi-MoC designs can be faithfully modeled and simulated. However, as the frameworks vary in the granularity of the MoC domains supported, one can imagine that the fidelity of the framework varies. For example, in Ptolemy II, an SDF model is untimed, and if a DSP application needs to compute over multiple cycles, it is less direct to model such a behavior in the SDF domain of Ptolemy II. Similarly, in the ForSyDe or similar denotational framework, if one wants to model rendez-vous style communication paradigm, it is usually not a part of the MoC classification in such frameworks. As a result, one has to approximate it by indirect means.

Another aspect of such frameworks is *expressiveness*. When a modeling framework does not impose particular structures or specific process constructors and combinators for modeling specific models of computation, but rather provides the user with the full expressiveness of a programming language to model systems according to the user's own modeling style and structure, the expressiveness is high. However, if the framework does not allow the user to make use of all possible ways that a standard programming language permits, but rather imposes structural and stylistic restrictions through process constructors and combinators, then such a framework has lower expressiveness than standard programming language. However, this lower expressiveness usually is matched with high fidelity, because according to this notion of expressiveness, a low expressiveness in a framework implies that it provides the user with strict structures and guidelines to faithfully reflect an intended theoretical model of computation, hence has high fidelity.

The question then arises as to what the compromises between high fidelity/lower expressiveness and low fidelity/high expressiveness are. The lower expressiveness often helps model designers to be structured and hence the modeling exercise is less prone to modeling errors. The higher expressiveness accompanied by full power of a standard programming language such as Java/C++, tends to lead users to be too creative in their model building exercise which may be a source of many errors. It

is our opinion that low expressiveness/high fidelity is the desired quality of a good modeling and simulation framework.

The term 'low expressiveness' may mislead readers to think that such frameworks are not capable of achieving much. But since we pair 'high fidelity' with low expressiveness, we insist that such frameworks can provide structures for modeling most common models of computation needed for an application domain. For example, Ptolemy II (if restricted only to use the existing MoCs, and not full expressive power of Java) is a low expressive/high fidelity framework for embedded software. However, ForSyDe in our view is low expressive/high fidelity for creating models with the purpose of abstraction and formal verification, but not necessarily for hardware/software co-design.

SystemC, as it is implemented in its reference implementation has a very low fidelity, because the only MoC that can be directly mapped to the current SystemC framework is the Discrete-Event MoC. The other MoCs needs to be indirectly mapped onto this particular MoC whose semantics is suited for digital hardware simulation, and it is based very much on VHDL simulation semantics. However, if SystemC is extended with specific MoC structures and simulation facilities, it has the possibility of becoming a very high fidelity modeling framework for simulation, and synthesis of embedded hardware/software systems.

Another important metric to compare different frameworks is the simulation efficiency of complex models in the framework. It turns out that this metric is again closely tied to the fidelity and expressiveness issue. If a framework is structured to support multi-MoC modeling and simulation, it can be appropriately curtailed for simulation efficiency by exploiting the MoC specific characteristics. For example, consider the current reference implementation of SystemC with single discrete-event MoC kernel. Whenever an SDF model is simulated on such a kernel, the static scheduling possibility of SDF models goes unexploited and hence simulation efficiency is much less, compared to an extended framework based on SystemC, where SDF specific structures, and simulation kernels are available. Therefore, fidelity plays an important role in determining the efficiency of simulation of models in a framework.

Heterogeneous and Hierarchical Modeling and multi-MoC Frameworks

Today's embedded systems, and system-on-chip architectures are heterogeneous in nature. The different components, hardware as well as software, are best modeled by different models of computation. Systems that interact with analog domains such as sensor systems are best modeled at least at the interfaces with a continuous time MoC, while a DSP

component may be modeled better with an SDF MoC, and a software component interacting with them may be best modeled with an untimed state machine model.

As a result, any framework for modeling such complex systems requires to have high fidelity, as per our discussion in the previous section. So the need for heterogeneity and support for multi-MoC modeling and simulation are unquestionable.

We however, feel that another important aspect of such modeling frameworks which is often ignored by designers is the ability to express hierarchy. This must be accompanied with a simulation facility that does not require to flatten the hierarchy into one large model while simulating a hierarchical model. In this book we do not get into the hierarchical modeling, since the main focus of this book is on heterogeneous modeling framework based on SystemC. Nevertheless, it is imperative to talk about hierarchy in brief, because in many of the design decisions we make, we keep the goal of extending the framework for hierarchy in mind.

Often times, hierarchy is thought of as synonymous to *structural hierarchy* where structural models belonging to the same MoC are embedded within each other. Such hierarchy can be modeled in most hardware description languages such as VHDL, Verilog etc. We, however, mean a less trivial hierarchy, which is *behavioral* hierarchy. When a model in a certain MoC can contain another model belonging to a different MoC domain, inside itself, and such nesting is possible to arbitrary depth, that provides true behavioral hierarchy. Such nesting should not break the model's semantics, nor should it break the simulation capability. In real systems, for example, a finite state machine, each state may actually require to carry out a computation which is best expressed with an SDF. It is also possible that a CSP module may require an FSM embedded inside it. Support of such embedding are there in Ptolemy II. We believe that a truly heterogeneous model of computation based on SystemC needs to provide this facility in order to be successfully used in designing a full fledged hardware/software system.

Epilogue to this Preface

At this point, we hope our long preface has been able to convince some readers that the task we undertook is important but it is arduous. What we provide in this book is not claimed to be a complete solution, but a step towards the right direction, in our opinion. We would request the reader to read the rest of the book with this in mind. We understand that the multi-kernel SystemC we present here is not SystemC, because by definition SystemC is what the current standard specifies. At the

time of writing this preface, the latest version of reference implementation and standard specification was SystemC-2.0.1, and the language reference manual for this version is available at the OSCI website [49]. Therefore, we caution the readers that our multi-MoC SystemC is not official SystemC, nor can it be called SystemC. It is a modeling framework built on top of standard SystemC, and so far, we have not made any efforts to feed it back to the SystemC standards committee. However, we will be interested in doing so, if this book convinces enough readers about the need for extending SystemC in the ways we prescribe here. We would like to hear directly from readers, and encourage them to email one of the authors their opinions, questions and comments.

Hiren D. Patel

Sandeep K. Shukla

Acknowledgments

We would like to thank Sankar Basu and Helen Gill from the National Science Foundation for their encouragement and support for building the FERMAT lab under the NSF CAREER award CCR-0237947 which served as the home for the project work reported in this book. A part of the support for this work also came from an NSF NGS grant 0204028. The graduate studies of Hiren D. Patel is also supported by an SRC grant from the SRC Integrated Systems Program. Thanks are due to Professor Rajesh Gupta from the University of California at San Diego whose continuous encouragement and collaboration helped us along the way in this project.

We would also like to thank Mark de Jongh from Kluwer Academic Publishers for working with us and helping to ensure a fast turnaround time for the book. We also thank Cindy Zitter from Kluwer, for all her timely help with administrative issues.

We are grateful to Grant E. Martin, Chief Scientist, Tensilica, for encouraging the project with his foreword for the book, and providing us with feedback as and when we requested.

The students of FERMAT Lab at Virginia Tech, especially Deepak Abraham Mathaikutty, Debayan Bhaduri, Syed Suhaib and Nicholas Uebel have helped the project in different ways. The graduate students of Computer Synthesis class at Virginia Tech in the Spring of 2003, and Spring of 2004 had to endure some of this research material in the course, and we thank them for their enthusiasm about this research.

We thank Edward A. Lee and Stephen Neundorffer from the Ptolemy II group at the University of California at Berkeley for their inputs on many occasions on the idea of heterogeneous modeling in SystemC. We also thank the Ptolemy II project as being a source of guidance for various nomenclatures and examples used in this project.

We are also grateful to John Sanguinetti and Andy Goodrich from FORTE Design Systems for expressing their interest and faith in this work.

Last but not the least, we thank our families and friends for being patient at our involvement in this project and our inability to spend more time with them during the hectic period of completing the book.

Chapter 1

INTRODUCTION

1. Motivation

The technological advances experienced in the last decade has initiated an increasing trend towards IP-integration based System-on-Chip (SoC) design. Much of the driving force for these technological advances is the increasing miniaturization of integrated circuits. Due to the economics of these technologies where electronic components are fabricated onto a piece of silicon, the cost and size have exponentially reduced while increasing the performance of these components. Dating back to the early 1960s where two-transistor chips depicted leading edge technology, design engineers might have had difficulty envisioning the 100 million transistor chip mark that was achieved in 2001 [7]. Moore's law indicates that this progress is expected to continue up to twenty more years from today [44]. The ability to miniaturize systems onto a single chip promotes the design of more complex and often heterogeneous systems in both form and function [12]. For example, systems are becoming increasingly dependent on the concept of hardware and software co-design where hardware and software components are designed together such that once the hardware is fabricated onto a piece of silicon, the co-designed software is guaranteed to work correctly. On the other hand, a single chip can also perform multiple functions, for example network processors handle regular processor functionality as well as network protocol functionality. Consequently, the gap between the product complexity and the engineering effort needed for its realization is increasing drastically, commonly known as the *productivity gap*, as shown in Figure 1. Many factors contribute to the productivity gap

such as the lack of design methodologies, modeling frameworks & tools, hardware & software co-design environments and so on. Due to the hindering productivity gap, industries experience an unaffordable increase in design time making it difficult to meet the time-to-market. In our efforts to manage complex designs while reducing the productivity gap, we specifically address modeling and simulation frameworks that are generally crucial in the design cycle. In particular, we attempt at raising the level of abstraction to achieve closure of this productivity gap. In this effort, we build on the modeling and simulation framework of SystemC [49].

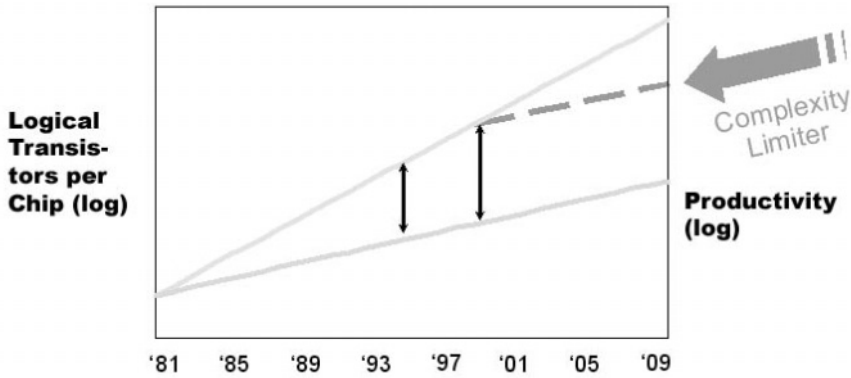


Figure 1.1. Productivity Gap [16]

2. System Level Design Languages and Frameworks

System Level Design Languages (SLDL) provide users with a collection of libraries of data types, kernels, and components accessible through either graphical or programmatic means to model systems and simulate the system behavior. A modeling framework provides a way to create a representation of models for simulation, and a simulation framework is a collection of core libraries that simulates a model.

In the past few years, we have seen the introduction of many SLDLs and frameworks such as SystemC, SpecC, SystemVerilog [49, 63, 65] etc., to manage the issue of complex designs. There also is an effort to lift the abstraction level in the hardware description languages, exemplified in the efforts to standardize SystemVerilog [65]. Another Java-based

system modeling framework developed by U. C. Berkeley is Ptolemy II [25]. The success of any of these languages/frameworks is predicated upon their successful adoption as a standard design entry language by the industry and the resulting closure of the *productivity and verification gaps*.

SystemC [49] is poised as one of the strong contenders for such a language. SystemC is an open-source SLDL that has a C++ based modeling environment. SystemC's advantage is that it has free simulation libraries to support the modeling framework. This provides designers with the ability to construct models of their systems and simulate them in a very VHDL and Verilog [69, 68] like manner. SystemC's framework is based on a simulation kernel which is essentially a scheduler. This kernel is responsible for simulating the model with a particular behavior. The current SystemC has a Discrete-Event based kernel that is very similar to the event-based VHDL simulator. However, the strive to attain a higher level of abstraction for closure of the productivity gap, while keeping the Discrete-Event (DE) simulation semantics are highly contradictory goals. In fact, most system models for SoCs are heterogeneous in nature, and encompass multiple Models of Computation (**MoC**) [37, 23] in its different components.

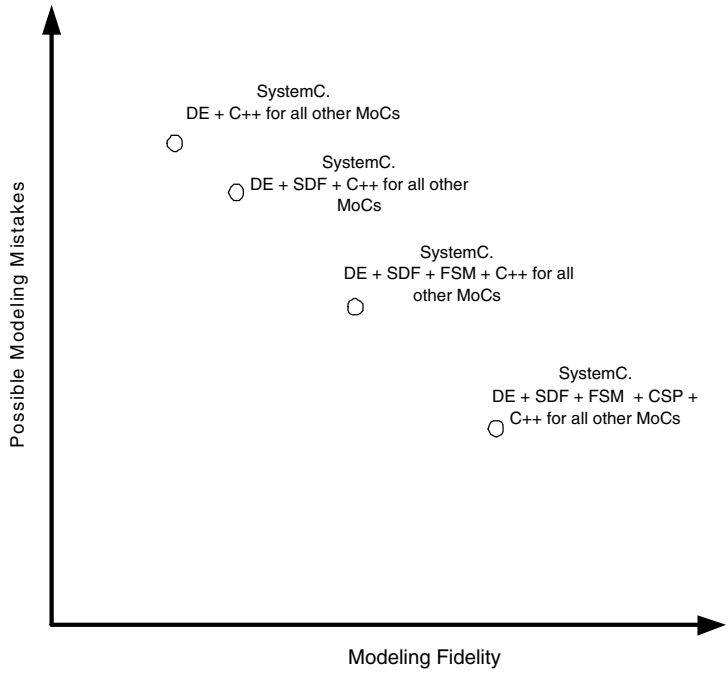


Figure 1.2. Possible Mistakes occurred versus Modeling Fidelity

Figure 1.2 displays the difficulty in being able to express MoC specific behaviors versus the possible modeling errors made by designers . Users restricted to DE semantics of SystemC lack in facilities to model other MoCs distinct from DE, requiring user-level manipulations for correct behavior of these MoCs. Due to these manipulations, designers may suffer from an increased number of modeling errors to achieve correct operation of a model. Hence, Figure 1.2 shows that SystemC with its DE kernel and functionality of C++ provides the most modeling expressiveness but also indicates more possible modeling mistakes. Conversely, the modeling expressiveness is significantly reduced when employing MoC-specific kernels for SystemC, but the possible errors made are also reduced.

We have therefore chosen the term 'fidelity' as a qualifying attribute for modeling frameworks. Restricting SystemC users to use only implemented MoC-specific structures and styles disallows users to use any feature afforded by free usage of C++, but such restricted framework offers higher fidelity. Fidelity here refers to the capability of the framework to faithfully model a theoretical MoC. It is necessary to think about simulation semantics of such a language away from the semantics of pure Discrete-Event based hardware models and instead the semantics should provide a way to express and simulate other Models of Computation. The inspiration of such a system is drawn upon the success of the Ptolemy II framework in specification and simulation of heterogeneous embedded software systems [25]. However, since SystemC is targeted to be the language of choice for semiconductor manufacturers as well as system designers, as opposed to Ptolemy II, its goals are more ambitious. Our focus is in developing an extension to the SystemC kernel to propose our extended SystemC as a possible heterogeneous SLDL. We demonstrate the approach by showing language extensions to SystemC for Synchronous Data Flow (SDF), Communicating Sequential Processes (CSP) and Finite State Machine (FSM) MoCs. A common use of SDF is in Digital Signal Processing (DSP) applications that require stream-based data models. CSP is used for exploring concurrency in models, and FSMs are used for controllers in hardware designs. Besides the primary objective of providing designers with a better structure in expressing these MoCs, the secondary objective is to gain simulation efficiency through modeling systems natural to their behavior.

2.1 Simulation Kernel and MoC

The responsibility of a simulation model is to capture the behavior of a system being modeled and the simulation kernel's responsibility is to simulate the correct behavior for that system. In essence, the Model of

Computation dictates the behavior which is realized by the simulation kernel. An MoC is a description mechanism that defines a set of rules to mimic a particular behavior [39, 37]. The described behavior is selected based on how suitable it is for the system. *Most MoCs describe how computation proceeds and the manner in which information is transferred between other communicating components as well as with other MoCs.* For more detailed discussion on MoCs, readers are referred to the preface of this book and [32, 25].

2.2 System Level Modeling and Simulation

For most design methodologies, simulation is the foremost stage in the validation cycle of a system, where designers can check for functional correctness, sufficient and appropriate communication interactions, and overall correctness of the conceptualization of the system. This stage is very important from which the entire product idea or concept is modeled and a working prototype of the system is produced. Not only does this suggest that the concept can be realized, but the process of designing simulation models also refines the design. Two major issues that need consideration when selecting an SLDL for modeling and simulation are *modeling fidelity* and *simulation efficiency*, respectively.

An SLDL's modeling fidelity refers to the constructs, language and design guidelines that facilitate in completely describing a system with various parts of the system expressed with the most appropriate MoCs. Some SLDLs make it difficult to construct particular models due to the lack of fidelity in the language. For example, SystemC is well suited for Discrete-Event based models but not necessarily for Synchronous Data Flow (SDF) models. [23, 45] suggest extra designer guidelines that are followed to construct Synchronous Data Flow models in SystemC. These guidelines are required because the SDF model is built with an underlying DE kernel, increasing the difficulty in expressing these types of models. This does not imply that designs other than DE-based systems cannot be modeled in SystemC, but indicates that expressing such models is more involved, requiring designer guidelines [23, 45].

Simulation efficiency is measured by the time taken to simulate a particular model. This is also a major issue because there are models that can take up to hours of execution time. For example, a PowerPC 750 architecture in SystemC [42] takes several hours to process certain testbenches [61]. With increased modeling fidelity through the addition of MoC kernels in SystemC, models for better simulation efficiency can be created.

Choosing the appropriate SLDL requires modeling frameworks to be appropriately *matched* to allow for meaningful designs. The simulation

framework also has to be *matched* to allow for correct and efficient validation of the system via simulation. The frameworks must provide sufficient behavior to represent the system under investigation and must also provide a natural modeling paradigm to ease designer involvement in construction of the model. Consequently, most industries develop proprietary modeling and simulation frameworks specific for their purpose, within which their product is modeled, simulated and validated. This makes simulation efficiency a gating factor in reducing the productivity gap. The proprietary solutions often lead to incompatibility when various parts of a model are reusable components purchased or imported as soft-IPs. Standardization of modeling languages and frameworks alleviate such interoperability issues.

To make the readers aware of the distinction between modeling guideline versus enforced modeling framework, we present a pictorial example of how SDF models are implemented in [23, 45] (Figure 1.3) using SystemC's DE kernel. This example shows a Finite Impulse Response (FIR) model with *sc_fifo* channels and each process is of *SC_THREAD()* type. This model employs the existing SystemC Discrete-Event kernel and requires handshaking between the Stimulus and FIR, and, FIR and Display blocks. The handshaking dictates which blocks execute, allowing the Stimulus block to prepare sufficient data for the FIR to perform its calculation followed by a handshake communication with the Display block.

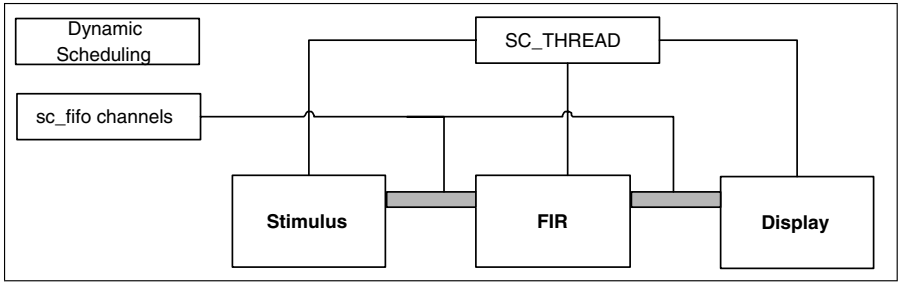


Figure 1.3. FIR model with Designer Guidelines [45, 23]

The same example with our SDF kernel is shown in Figure 1.4. This model uses *SDFport*s specifically created for this MoC for data passing instead of *sc_fifo*s and no synchronization is required since *static scheduling* is used. The model with the SDF kernel abandons any need for handshaking communication between the blocks and uses a scheduling algorithm at initialization time to determine the *execution schedule* of the blocks.

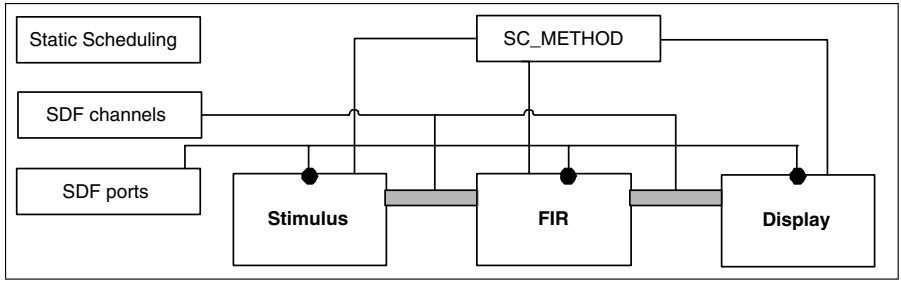


Figure 1.4. FIR model with our SDF kernel

We provide further explanation about these modeling examples in Chapters 3 and 5.

2.3 Simulation Efficiency Versus Fidelity of MoCs

It is conceivable that there can be large, detailed models that take enormous amounts of time for simulation [61]. We measure simulation performance in terms of the amount of time taken to simulate a model from start to end. We believe that matching components to models with the most appropriate MoC enhances simulation efficiency.

An MoC describes the underlying behavior of a system on top of which a model following that particular behavior is simulated. The systems expected behavior must be modeled appropriately by the modeling framework, otherwise an unnatural modeling scheme has to be used to impose the behavior and the simulation efficiency will almost always suffer. When we describe a simulation framework supporting the Model of Computation, we particularly refer to kernel-support for that MoC. Our motivation involves modeling and simulating SDF, CSP, FSM and other MoC designs in SystemC by introducing distinct simulation kernels for each MoC. Although meant to be a system level language, the current SystemC simulation kernel uses only a non-deterministic DE [39, 37, 49] kernel, which does not explicitly simulate SDF, CSP or FSM models in the way that they could be simulated more efficiently and easily.

In other words, current SystemC is very well suited for designs at the register transfer level of abstraction, similar to VHDL or Verilog models. However, since SystemC is also planned as a language at higher levels of abstraction, notably at the system level, such Discrete-Event based simulation is not the most efficient. An SoC design might consist of DSP cores, microprocessor models, bus models etc. Each of these may be most naturally expressible in their appropriate MoCs. To simulate such a model the kernel must support each of the MoCs interlaced in the

user model. This allows designers to put together heterogeneous models without worrying about the target simulation kernel and discovering methodologies to enforce other MoCs onto a single MoC kernel. Figure 1.5 shows an example of a digital analyzer [32] that employs two MoCs, one being an untimed Data Flow MoC and the other being a timed model based on the DE MoC. The Controller IP and Analyzer IP could be possibly obtained from different sources requiring the interfaces to allow for communication between the two. Support for modeling these IPs requires the simulation framework to provide the underlying MoCs, otherwise programmatic alterations are required to the original IPs for conformity. Again, it is crucial for modeling frameworks to allow designers to express their models in a simple and natural fashion. This can be achieved by implementing MoC-specific kernels for the modeling framework along with sufficient designer guidelines in constructing models.

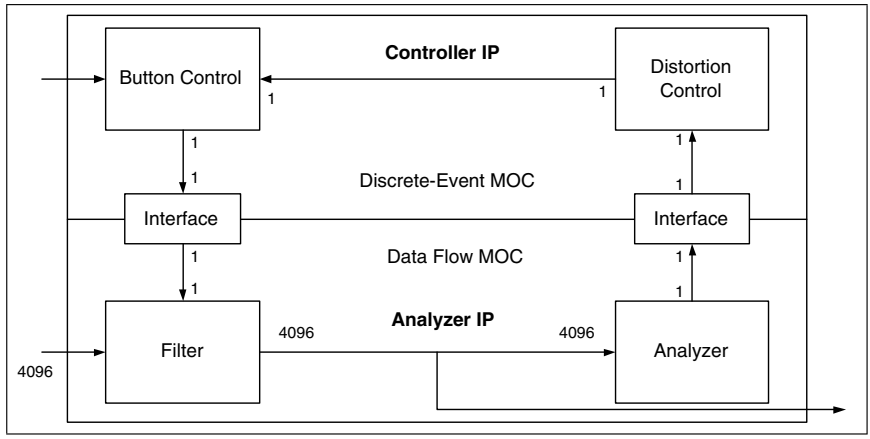


Figure 1.5. Example of Digital Analyzer using Models of Computation [32]

3. Our Approach to Heterogeneous Modeling in SystemC

Heterogeneous modeling in SystemC has been attempted before, but has only been attained by using designer guidelines [23, 45]. One of the first attempts at designer guidelines for heterogeneous modeling is provided in [23]. We find these guideline based approaches to be quite unsatisfactory. In [23] the authors model SDF systems using *SC_THREAD()* processes and blocking read and write *sc_fifo* channels. This scheme uses dynamic scheduling at runtime as opposed to possible static scheduling of SDF models. They accommodate an SDF model with the underlying

DE kernel. We argue that this is not an efficient method of implementing SDF in SystemC nor is it natural. The adder example they present [23] uses blocking *sc_fifo* type channels that generate events on every blocking read and write resulting in an increase in the number of delta cycles. Furthermore, the use of *SC_THREAD()* introduces context-switching overhead further reducing the simulation efficiency [20]. Although [23] promotes that tools may be designed to exploit static scheduling algorithms to make efficient SDF simulation in systems, no solution or change in the kernel was provided. They only hint at some source level transformation techniques in enabling SDF systems with the underlying DE reference implementation kernel.

Another effort in modeling Data Flow systems in SystemC was done in [45], where they provide an Orthogonal Frequency Division Multiplex (OFDM) example as an SDF model. However, this model does not exploit the SDF Model of Computation either since this example also uses blocking read and write resulting in dynamic scheduling. Once again this approach defeats the purpose of modeling SDF systems because once recognized as SDF, they should be statically scheduled. These models presented in [23, 45] can be converted such that there is no need for *SC_THREAD()* or *SC_CTHREAD()* processes, communication signals to pass control, and most importantly dynamic scheduling. We present an extension of SystemC in which an SDF kernel interoperable [14] with the existing DE kernel can eliminate the need for dynamic scheduling and designer guidelines specific for SDF models. Similar extensions are also prototyped for CSP and FSM Models of Computation.

3.1 Why SystemC?

In addition, the preset C++ language infrastructure and object- oriented nature of C++ used to define SystemC extends usability further than any existing hardware description language (HDL) at present. SystemC provides embedded system designers with a freely available modeling and simulation environment requiring only a C++ compiler. We have undertaken the task of building a SystemC simulation kernel that is interoperable with the existing simulation kernel of the reference implementation, but which employs different simulation kernel functions depending on which part of the design is being simulated. In this book, we focus on building the simulation kernel appropriate for the SDF [37, 39, 25], CSP [27] and FSM models.

Once kernel specific MoCs are developed, they can be used as standalone for a specific system and even in unison to build heterogeneous models. Communication between different kernels could either be event-driven or cycle-driven depending on the choice of the implementer. Event-

driven communication describes a system based on message passing in the form of events, where events trigger specific kernels to execute. As an example, our SDF kernel executes only when an event from the DE kernel is received. On the other hand cycle-driven refers to a system whereby for a number of cycles one particular kernel is executed followed by another.

Our goal is not to provide a complete solution, but to illustrate one way of extending SystemC. We hope this would convince the SystemC community to build industry strength tools that support heterogeneous modeling environments. However, the standardization of SystemC language is still underway, but as for the kernel specifications and implementations, SystemC only provides the Discrete-Event simulation kernel. We build upon the current SystemC standard in laying a foundation for an open source multi-domain modeling framework based on the C++ programming language which has the following advantages:

- The open-source nature of SystemC allows alterations to the original source.
- Object- oriented approach allows modular and object- oriented design of SystemC and its extensions to support the SDF kernel.
- C++ is a well accepted and commonly used language by industry and academics, hence making this implementation more valuable to users.

4. Main Contributions of this Book

We extend the existing SystemC modeling and simulation framework by implementing a number of MoC-specific kernels, such as SDF, CSP and FSM. Our kernels are interoperable with SystemC's DE kernel allowing designers to model and simulate heterogeneous models made from a collection of DE and SDF models or other combinations corresponding to different parts of a system. By extending the simulation kernel, we also improve the modeling fidelity of SystemC to naturally construct models for SDF, CSP and FSM. Furthermore, we increase the simulation efficiency of SDF and DE-SDF models in SystemC. We do not benchmark for the other kernels, but we believe they will also show similar improvements.

Synchronous Data Flow MoC is by no means a new idea on its own and extensive work has been done in that area. [5] shows work on scheduling SDFs, buffer size minimization and other code optimization techniques. We employ techniques presented in that work, such as their scheduling algorithm for SDFs. However, the focus of [5] pertains to

synthesis of embedded C code from SDF specifications. This allows their SDF representations to be abstract, expressing only the nodes, their connections, and token values. This is unlike the interaction we have to consider when investigating the simulation kernel for SystemC. We need to manipulate the kernel upon computing the schedules requiring us to discuss the Data Flow theory and scheduling mechanisms. Our aim is not in just providing evidence to show that changes made in the kernel to accommodate different MoCs improves simulation efficiency, instead, we aim at introducing a heterogeneous modeling and simulation framework in SystemC that does also result in simulation efficiency.

Hoare's [27] Communicating Sequential Processes is an MoC for modeling concurrent systems with a rendez-vous style synchronization between processes. Rendez-vous occurs between two processes ready to communicate. Otherwise, the process attempting the communication suspends itself and can only be resumed once the process receiving the transfer of data from that process is ready to communicate. We implement the CSP kernel with SystemC's coroutine packages used to create thread processes. CSP processes in our CSP kernel are SystemC thread processes with a different scheduling mechanism suitable for CSP modeling.

Most hardware systems have controllers modeled as FSMs. The current SystemC can model FSMs using standard switch statements, but the FSM kernel provides a direct method of invoking different states. Combining SDF, CSP and FSM MoCs to create heterogeneous models shows the usefulness of our multi-MoC SystemC framework. The famous Dining Philosophers problem has an elegant solution that can be modeled in CSP. Furthermore, a deadlock avoidance technique can be added using an FSM controller as a footman assigning seats. We discuss this model in further detail in Chapter 9.

Further improvements in synthesis tools such as Cynthesizer and Cocentric SystemC Compiler [17, 64] will provide designers with the capability of effective high level synthesis. We hope to initiate the process by which SystemC can fulfill its purpose of being a high level and multi-domain modeling framework for industry use. We plan on distributing our prototype implementation that introduces heterogeneity in SystemC to the SystemC community via a freely downloadable distribution [36].

Chapter 2

BACKGROUND MATERIAL

1. System Level Modeling and Simulation Methods

Numerous system level modeling and simulation frameworks are developed for specific purposes and systems. Examples of recently introduced system level modeling languages are SpecC, SystemC, SystemVerilog, etc. [63, 49, 65]. Among the other frameworks for system level modeling, Ptolemy II [25] is the most well known. Another breed of system level modeling frameworks consists of Opnet, and NS-2 which are network oriented modeling and simulation frameworks [48, 46]. These system level design languages and frameworks are usually application-specific. For example, tools like NS-2 and Opnet are predominantly used to create network models and simulate network protocols whereas VHDL [69] for RTL designs and SystemC for RTL designs and system level designs. Likewise, Ptolemy II is designed to model heterogeneous embedded systems with a specific focus on embedded software synthesis.

Some of these allow designers to create models either through graphical interfaces such as in Ptolemy and Opnet [48], or programmatic means as in SystemC or VHDL [49, 69]. Since these languages are application-specific, their expressiveness is limited to the particular targeted systems. The simulation frameworks for these languages also differ in their implementations where some are event-driven (such as Ptolemy II and SystemC) and others are cycle-based. SystemC being a library of C++ classes, promotes object based programming and in turn promotes object (or module) based modeling. This allows models to maintain a much higher level of abstraction than simply RTL models making it appropriate for system level designs. Current Verilog [68] on the other hand does

not allow flexibility in creating an object oriented design and limits itself to mainly RTL and block level models.

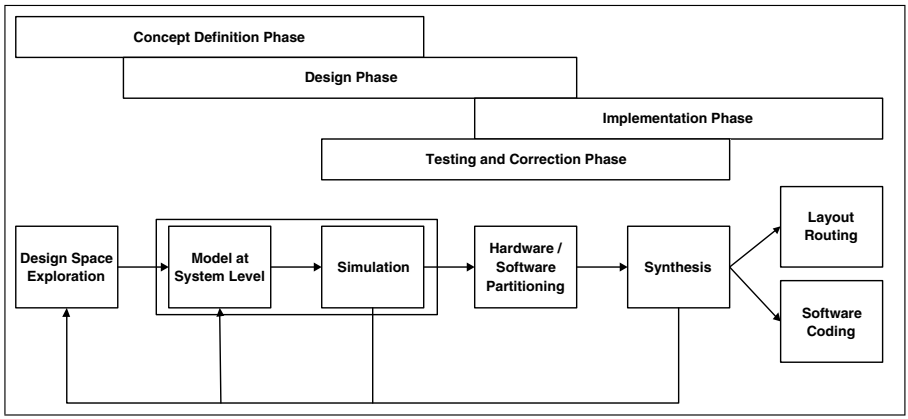


Figure 2.1. Design Cycle

Figure 2.1 shows that simulation is an important stage in the design cycle of a system. The purpose of simulation is validation of the system such that design errors are captured early in the design cycle. Ample time is spent in this design phase making simulation efficiency a gating factor in reaching the time-to-market goals.

2. Models of Computation and Heterogeneous Modeling at System Level

We suggest that an emphasis on improving simulation efficiency without providing kernel-support for different MoCs is not possible through mapping of their behaviors onto a specific kernel for a simulation framework. Currently, models which are best represented with different MoCs can be modeled and simulated in SystemC with a single Discrete-Event kernel, by mapping any Model of Computation onto this kernel. However, the cost is paid in the simulation efficiency. One can see this from the examples in [23, 45] for the SDF MoC. The lack of fidelity in current SystemC for MoCs other than Discrete-Event, makes it difficult to model these MoCs. Modeling at a higher level of abstraction requires the kernel to accommodate the "abstracted system" rather than having the designer to force the abstracted system on to the DE kernel. In this sense, SystemC is not yet a heterogeneous modeling and simulation framework, lacking the expressiveness to simulate MoCs other than the DE MoC. Our focus is on providing design ease through better fidelity and simulation efficiency in SystemC.

Having a brief understanding of the need for Models Of Computations as appropriate and suitable descriptions of systems, it is necessary to understand how the evolution of MoCs impacts design projects. The accepted practice in system design, guides the construction of the hardware parts first leaving the software components to be designed later [37]. Several design iterations are needed during the software design stage. Hence, design methodologies to create hardware and software components that are correct by construction and aid in reducing design time are sought. Recent efforts to address this challenge are devoted to hardware & software co-design resulting to the design terminology coined "function-architecture co-design" [37], where they argue that the key problem is not of hardware & software co-design but of

"the sequence consisting of specifying what the system is intended to do with no bias towards implementation, of the initial functional design, its analysis to determine whether the functional design satisfied the specification, the mapping of this design to a candidate architecture, and the subsequent performance evaluation" [37].

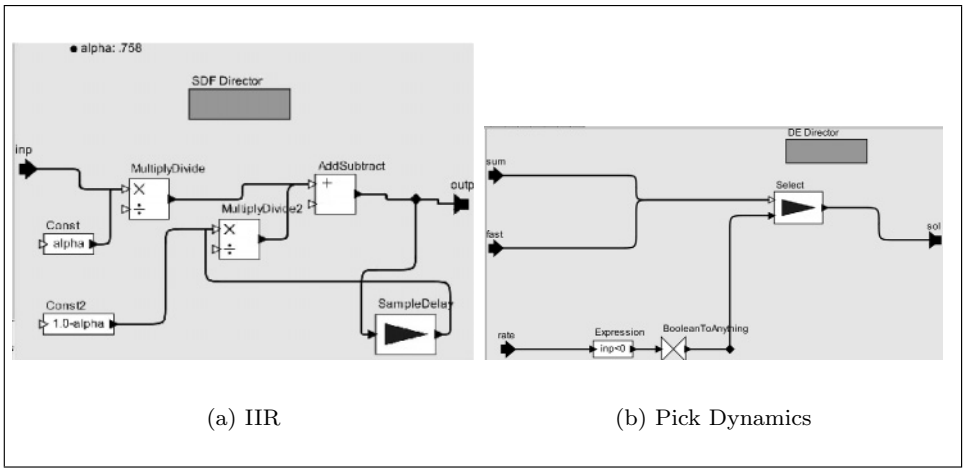
We understand the problem as the lack of fidelity in the frameworks and languages to express different Models of Computation for different parts of a large system. With multiple MoCs, designers can model systems more natural to the specific function. For example, DSP applications work on a stream passing which is akin to token flow paradigm that is well suited for a Data Flow (DF) MoC or sometimes a more specialized DF called the Synchronous Data Flow. A token is an encapsulation of data being transferred through the system. Providing any System Level Design Language with enough fidelity to express different MoCs accurately relaxes the modeling paradigm and having kernel-support for MoCs, simulation performance also increases.

3. Ptolemy II: A Heterogeneous Modeling and Simulation Framework

A specific framework that does encompass the "function-architecture co-design" principle is Ptolemy II [25]. Ptolemy II is an open-source and Java-based design language. Ptolemy's major focus is on embedded systems, in particular on systems that employ mixed technologies such as hardware and software, analog and digital, etc.

The models are constructed with different MoCs dictating the behavior of different parts of a system [6]. Ptolemy II introduces *directors* that encapsulate the behavior of the MoCs and simulate the model according to it. Behavior of these MoCs efficiently handle concurrency

and timing since these are integral requirements for embedded systems that need simultaneous and concurrent operations. Every *director* provides a notion of a *domain* in Ptolemy II. To mention a few of them, Ptolemy II has DE and SDF directors that can impart the Discrete-Event and Synchronous Data Flow behaviors respectively to the models in their domains. A component-based design approach is used where models are constructed as a collection of interacting components called *actors*. Figure 2.2 shows a diagram depicting the actor component and also displaying the interaction of a collection of components. Actors in



(a) IIR (b) Pick Dynamics

Figure 2.2. Composite Actors for Eye Model in Ptolemy II [3]

Ptolemy II can communicate with each other and execute concurrently. These actors require a medium of communication to other actors introducing the idea of channels. Channels pass data from ports of one actor to the connected ports of another actor (or sometimes to itself). Using channels implies that actors can not necessarily interfere with each other since they can only interact with other actors through the channels. Figure 2.2 shows actors, ports and other elements of composite actors in a Ptolemy model. These are two of the sub-models of a larger model for the biological feedback system modeling the functions of an Eye [3]. Ptolemy II also supports hierarchical actors that can contain actors within itself and connect to other actors through external ports.

Ptolemy II supports Models of Computation as their underlying behavior descriptors for their *director*s. Since Ptolemy II is geared towards embedded systems, the MoCs in Ptolemy II are particular to embedded software synthesis. A brief definition of some of the MoCs are presented

below taken from [6]. For a full list of MoCs implemented as domains in Ptolemy II, please refer to [6].

3.1 Component Interaction (CI)

The basis of the Component Interaction (CI) domain follows the construct for a data-driven or demand-driven application. An example of data-driven application is an airport departure and arrival display where information about flight departures, arrivals, delays, times, number, etc. are displayed for the public. Whenever a status changes of a particular flight, changes are made on the main server raising an event signaling the displays to perform a refresh or update.

Demand-driven computation on the other hand refers to the more common web-browsing environment where an HTTP request to a server acts as the demand resulting in an event on the server for a webpage. This event causes computation to return the requested webpage (given sufficient privilege for access) to the client requesting the webpage through a network protocol such as TCP.

3.2 Communicating Sequential Processes (CSP)

In the CSP domain, actors are concurrently executing processes. The communication between two or more CSP actors is carried out by atomic and instantaneous actions called rendez-vous [6, 27]. If a process is ready to communicate with another process, then the ready-to-communicate process is stalled until the other process is ready to communicate. Once the processes are both ready the communication occurs simultaneously in one non-interruptible quantum, hence atomic.

3.3 Continuous Time (CT)

As the name suggests, in the Continuous Time (CT) domain, the interaction between actors is via continuous-time signals. The actors specify either linear or nonlinear algebraic/differential equations between their inputs and outputs. The director finds a fixed point in terms of a set of continuous-time functions that satisfy the equations.

3.4 Discrete-Events (DE)

In the Discrete Event (DE) domain, actors communicate through events on a real time line. An event occurs on a time stamp and contains a value. The actors can either fire functions in response to an event or react to these events. This is a common and popular domain used in specifying digital hardware. Ptolemy provides deterministic semantics

to simultaneous events by analyzing the graph for data dependence to discover the order of execution for simultaneous events.

3.5 Finite-State Machine (FSM)

The FSM domain differs from the actor-oriented concept in Ptolemy II in that the entities in the domain are represented by states and not actors. The communication occurs through transitions between the states. Furthermore, the transitions can have guard conditions that dictate when the transition from one state to the other can occur. This domain is usually useful for specifying control state machines.

3.6 Process Networks (PN)

In the Process Network domain, processes communicate via message passing through channels that buffer the messages. This implies that the receiver does not have to be ready to receive for the sender to send, unlike the CSP domain. The buffers are assumed to be of infinite size. The PN MoC defines the arcs as a sequence of tokens and the entities as function blocks that map the input to the output through a certain function. This domain encapsulates the semantics of Kahn Process Networks [34].

3.7 Data Flow (DF)

DF domain is a special case of Process Networks where the atomic actors are triggered when input data is available. Otherwise data is queued.

3.8 Synchronous Data Flow (SDF)

The Synchronous Data Flow (SDF) domain, operates on streams of data very suitable for DSP applications. SDF is a sub-class of DF except that the actor's consumption and production rates dictate the execution of the model. The actors are only fired when they receive the number of tokens defined as their consumption rate and expunge similarly only the number of tokens specified by the production rate. Moreover, static scheduling of the actors is performed to determine the schedule in which to execute the actors.

Ptolemy II's open source nature allows experienced Java programmers to add their own directors as desired. However, the domains that are implemented provide a sufficient base for the targeted embedded systems that Ptolemy is designed for. Having the ability to create personalized MoCs provides a very extensible and flexible modeling and simulation framework. Furthermore, Java is a purely object oriented programming language with a large volume of Graphical User Interface (GUI) libraries.

Although SystemC does not have a mature GUI for system level modeling besides the efforts in Cocentric System Studio [64], we show in this book that SystemC can also be extended to support MoCs through appropriate kernel development.

4. SystemC: Language and Framework

SystemC is also an open-source initiative [49] that provides a programmatic modeling framework and a Discrete-Event based simulation framework. In SystemC, the notion of a component or an actor *a la* Ptolemy is simply an object module constructed by the macro *SC_MODULE(...)* [Listing 2.1, Line 3]. This module must be of a process type that is defined in its constructor *SC_CTOR(...)* [Listing 2.1, Line 11]. An example of the source is in Listing 2.1.

Listing 2.1. SC_MODULE example

```
1 #include <systemc.h>
2
3 SC_MODULE(__module_example__) {
4
5   /* Signals and Port declarations */
6   sc_in_clk clock_in;
7
8   sc_in<bool> happy_port;
9   s_out<bool> sad_port;
10
11  SC_CTOR(__module_example__) {
12    /* Can have the following processes */
13    // SC_CTHREAD(...)
14    // SC_THREAD(...)
15    // SC_METHOD(...)
16    SC_THREAD(entry)  {
17      /* Sensitivity List */
18      sensitive << clock_in.pos();
19    };/* END SC_THREAD */
20  }/* END SC_CTOR */
21
22  void entry();
23 };/* END SC_MODULE */
```

Options for the type of processes that can be used in SystemC are:

***SC_METHOD(...)*:** *SC_METHOD()* processes execute from start to end and are triggered by their sensitivity list.

***SC_THREAD(...)*:** *SC_THREAD()* processes are light-weight threads created via the quick-thread package for C++ [35]. These processes are usually programmed as an infinite loop with suspension points using *wait(...)* statements. *SC_THREAD()*s are spawned once and continue to execute based on the infinite loop and suspension points.

***SC_CTHREAD(...)*:** *SC_CTHREAD()* are derived from *SC_THREAD()* processes with the addition that the *SC_CTHREAD()* resumes execution after a suspension point at least every chosen clock pulse. The constructs *wait(...)* and *wait_until(...)* can be used to suspend a process. *wait_until(...)* provides a mechanism for dynamic sensitivity in SystemC. However, *SC_CTHREAD()* and *wait_until(...)* will be deprecated in the newer version of SystemC.

The *SC_CTOR(...)* calls the constructor for that module that can create a module of the above three types. The sensitivity list dictates when the module is to execute upon events on signals/channels. Every primitive signal and channel generates events to indicate that the particular signal/channel has been updated. When events occur on signals or channels on the sensitivity list, the process is expected to fire. In the case of *SC_CTHREAD()* processes, it is necessary to define a clock edge on which the thread will execute. A user can define more than one kind of process and bind it with its appropriate *entry* function [Listing 2.1, Line 22]. [Listing 2.1, Line 16] binds *entry* to a *SC_THREAD()* process. This means that one instance of *SC_MODULE(...)* can contain more than one *entry* functions of varying process types. These *entry* functions behave according to the process bound with it.

Every *SC_MODULE(...)* can have port declarations such as *happy_port* and *sad_port* [Listing 2.1, Line 8 - 9]. Channels/signals are used to connect one module to another via these ports. Input ports such as *sc_in*<...> receive input from the signals and output ports *sc_out*<...> transmit data onto the signals. The ports like the signals are of template type allowing for types to vary. The main top-level module has to create signals using the *sc_signal*<...> declaration and these must be connected with the corresponding module ports. During the evolution of SystemC, design alterations in terms of functionality, semantics and syntax are expected. For starters, the newer version of SystemC expects deprecating the *SC_CTHREAD()* process entirely, along with *wait_until()* function calls. We are aware of these expected changes, but specifically adhere to the SystemC standards set by version 2.0.1.

SystemC does not provide a graphical user interface from which one can construct models, though SystemC Cocentric Studio [64] is an attempt at performing this. When comparing to Ptolemy II, a module can be related to an actor, the channels as signals or channels in SystemC, and directors as the SystemC kernel. Since SystemC extends the C++ language using classes, macros and such C++ methodologies, almost any functionality can be added to SystemC models allowing construction of most MoCs. However, at this point, this comparison is only valid when

referring to a particular MoC, the Discrete-Event MoC. This is simply because SystemC kernel does not support any other MoC in their kernel other than the Discrete-Event Model of Computation. The main element that defines this MoC is the commonly used Evaluate-Update paradigm that will be discussed in the next chapter. Nonetheless, this brief explanation of SystemC should provide the reader enough insight into the basic modeling framework employed particular to the DE MoC. We continue to explain in detail the Discrete-Event kernel in Chapter 3.

5. Implemented Models of Computation

In this book, we present an avenue through which SystemC can be made a multi-domain heterogeneous modeling and simulation framework by introducing the Synchronous Data Flow (SDF), Communicating Sequential Processes (CSP) and Finite State Machine (FSM) Models of Computation kernels in SystemC. Please note that our implementation is a reference implementation. An industry strength design of heterogeneous kernels which support behavioral hierarchy is beyond our current scope. In the following subsections we briefly describe with illustrative examples each MoC we have implemented, namely SDF, CSP and FSM.

5.1 Synchronous Data Flow in SystemC

We present an example of a simple SDF model for an image encoder from [24]. The block diagram is shown in Figure 2.3. This image processing example takes in an image matrix and partitions it into four by two matrices that are then run through a vector quantizer which is then reassembled in the end into a compressed image. This procedure is repeated until the source matrix is empty.

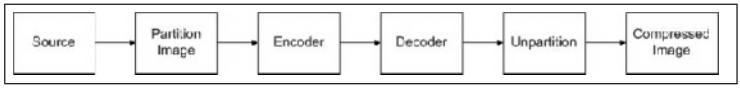


Figure 2.3. Image Processing Example

This example in Figure 2.3 is suitable for the SDF paradigm because each function block should only execute when it has enough inputs on its arcs. This is an ideal situation for static scheduling of the model. Current SystemC DE kernel does not allow static scheduling of a system like Figure 2.3. However, modeling or designer guidelines can be used to model such a system. This enforces an SDF-like system onto a DE kernel reducing the simulation efficiency of the model.

Sacrificing simulation efficiency by using designer guidelines might be an overstatement in that many designers are not aware or concerned about the simulation degradation incurred when simulating models which are not naturally a candidate for the Discrete-Event MoC implemented in SystemC's kernel. However, there has been some discussion and work in providing designer guidelines using the existing expressiveness in SystemC. For SDF, [23, 45] suggest guidelines that they term the *golden recipe* describing the use of *SC_THREAD()* processes with a communication medium of *sc_fifo* channels. Since these channels are used with *SC_THREAD()* processes, the channels must use blocking *read()* and *write()* functions to retrieve and put tokens onto the channels. They go on to describe mechanisms for stopping simulation that involve:

- Simulating the model for a finite amount of time using *sc_start(n)* where n is a time unit other than -1. This method has a constraint in that there must exist at least one timed model for the simulation to terminate.
- Return from *SC_THREAD()* processes stalling the simulation due to the lack of events for the kernel to process.
- Use the *sc_stop()* function call to terminate the simulation when a certain sentinel value is reached.

Here, we elaborate on the *golden recipe* by using the adder example in [23] to illustrate their inadequacy.

Listing 2.2. adder example

```
1 template <class T> SC_MODULE(DF_Adder)  {
2
3   sc_fifo_in<T> input1, input2;
4   sc_fifo_out<T> output;
5
6   void process()  {
7     while(1) {
8       output.write(input1.read() + input2.read());
9     }/* END WHILE */
10  }/* END PROCESS */
11
12  SC_CTOR(DF_Adder) {
13    SC_THREAD(process);
14  }/* END SC_CTOR */
15 }
```

Notice that in this example there are no *wait()* calls even though this is a *SC_THREAD()* process. Interestingly, this does not breach the modeling paradigm due to the use of *sc_fifo* channels [Listing 2.2, Line 3 - 4]. Synchronization need not be explicit in SDF models when using

SC_THREAD() processes due to the blocking *read()* and *write()* function calls. These blocking functions generate events for the SystemC DE kernel that need to be updated. This also means that the simulation requires no notion of a clock and completes using delta cycles. A delta cycle is a small step of time within the simulation time that does not advance the simulation time. This imposes a partial order of simultaneous actions [23]. However, these guidelines in [23, 45] are inadequate for a heterogeneous simulation framework. In [23] they model SDF systems using *SC_THREAD()* processes and blocking read and write *sc_fifo* channels [Listing 2.1, Line 8]. This scheme uses dynamic scheduling at runtime opposed to the conventional static scheduling of SDF models. They accommodate an SDF model with the underlying SystemC's DE kernel. We argue that this is not an efficient method of implementing SDF in SystemC nor is it natural. They also acknowledge the deficiency in simulation using this approach due to the increase in the number of delta cycles incurred by blocking *read()* and *write()* function calls. Furthermore, the use of *SC_THREAD()* introduces context-switching overhead further reducing the simulation efficiency [20]. Context-switching overhead refers to the cost of saving the state of the process and re-establishing when the process is to be executed again. Although [23] promotes that tools may be designed to exploit static scheduling algorithms to make efficient SDF simulation in systems, no solution or change in the kernel was provided. They only speculate some source level transformation techniques to model SDF systems with the underlying DE reference implementation kernel which we believe to be inadequate for design of complex systems.

We believe that these models presented in [23, 45] can be converted such that there is no need for *SC_THREAD()* or *SC_CTHREAD()* processes, communication signals to pass control, and most importantly dynamic scheduling. We present a novel concept of an SDF simulation kernel in SystemC, interoperable with the existing DE kernel eliminating the need for dynamic scheduling specific for SDF models. We present the SDF paradigm in further detail in Chapter 5.

5.2 Communicating Sequential Processes in SystemC

To illustrate the "Communicating Sequential Processes" MoC, we take the classical *Dining Philosopher's* problem cited in [27]. The problem is defined as follows: there are five philosophers PHIL0, PHIL1, PHIL2, PHIL3, and PHIL4, there is one dining room in which all of them can eat, and each one has their own separate room in which they spend most of their time thinking. In the dining room there is a round table with

five chairs assigned to each individual philosopher and five forks down on the table. In the middle of the table is a big spaghetti bowl that is replenished continuously (so there is no shortage of spaghetti).

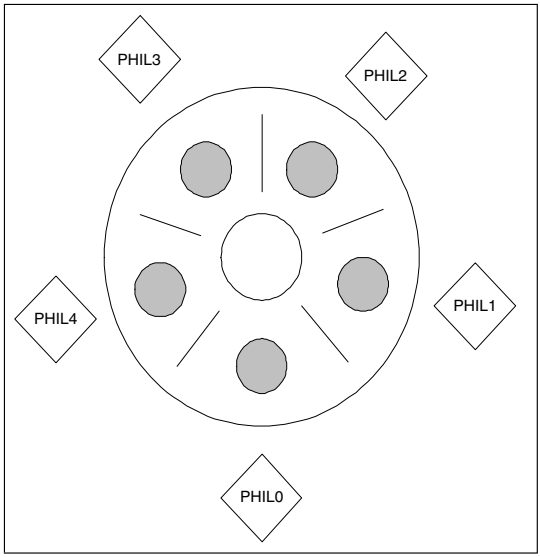


Figure 2.4. Dining Philosopher Problem

The duty of the philosophers is to spend most of their time thinking, but when hungry they were to enter the dining room, sit on their designated chair, pick up their left fork then pick up their right fork and begin eating. Since a philosopher requires two forks to eat the spaghetti, another philosopher requesting for a fork that is unavailable must simply wait until it is put down. Once done eating, the right fork is dropped, then the left and then the philosophers go back to their respective rooms to think. This classical example is tackled using threading and mutexes many times, however our approach is to allow SystemC to model such an example. Figure 2.4 displays the seating assignments of the philosophers. It also shows the dining room with the spaghetti bowl, the forks, and the assigned chairs. Though it may seem awkward to propose that inanimate objects have behavior, the forks in the Dining Philosopher model are processes along with the philosophers themselves. Therefore, there are a total of ten processes. The forks have distinct behaviors or realizing whether they are down on the table or whether they are not and the philosophers exhibit the behavior of thinking, picking up the left fork followed by the right, eating, then standing up to leave to think again. Figure 2.5 shows the processes with their communicating channels. We

return to this example later once we establish the design methodology for CSP in SystemC along with the details of the implementation.

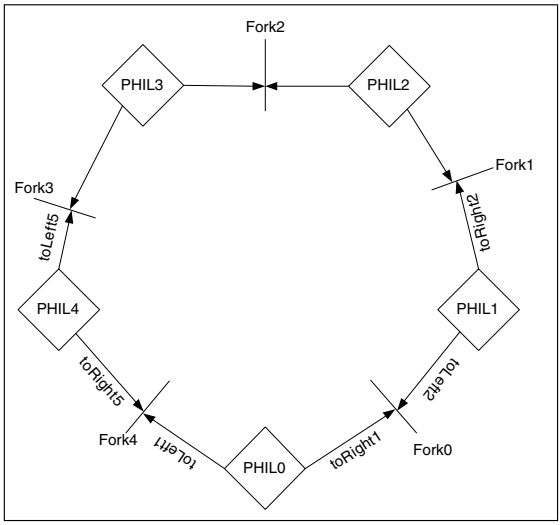


Figure 2.5. CSP Implementation of Dining Philosopher

We detail the CSP Model of Computation further in Chapter 6 as well as integration details to allow invocation of the reference implementation kernel through a CSP process.

Dining Philosopher Example in CSP Notation

In order to provide clarity to this recurring example, we present the *Dining Philosophers* solution in the CSP [28] notation. CSP is represented as a process algebra, and hence each process is a term in the algebra. Here we informally describe the operators of CSP and refer the readers to [28] for a detailed syntax and semantics of CSP.

In CSP, a sequential process can be described with a sequencing operator. This operator prefixes an *atomic action* a to a process term b. The new process constructed represents a process that engages in action a first, and then like process P. For example

$$Q = a \rightarrow P$$

means that process Q executes action a and then behaves like a process represented by the process term P.

One can also represent *non-deterministic choice* between two behaviors with a *choice* operator written as |. For example

$$Q = a \rightarrow P \mid b \rightarrow R$$

is used to represent a process Q, which behaves in two different ways based on which atomic action it can engage in, a or b. If it can engage in a first, then it executes action a first, and then behave like the process represented by the process term P, else it engages in b and then behaves like the process represented by the process term R. If both a and b are possible actions that it can engage in, then it non-deterministically chooses one of the two behavioral courses. One can naturally extend the choice operator from binary to n-ary, so a process can have more than two non-deterministic choices and courses of actions. Often times if the action set is indexed, for example if $a[i]$ such that $i = 0..n$, and the behaviors following action $a[i]$ is describable by a process term also indexed by i, say $P[i]$, then a compact representation of the choice is written as

$$Q =|_{i=0..n} \ (a[i] \rightarrow P[i])$$

Although the original CSP has no direct syntax to describe *priority* among the possible choices, later on variants of CSP have been proposed to encode priority among the choices. Priority can be partially ordered in general, but for our purposes we assume total order on the list of choices in terms of priority. So we will use a $\rightarrow$ on top of the choice operator $|$, and write it as $\overrightarrow{|}$ to denote prioritized choice with the direction of priority being decreasing from left to right in the term description.

To represent indefinite repetition of a sequential behavior, often written with loops in standard programming notation, CSP allows use of recursive definitions of process terms. For example

$$Q = a \rightarrow Q$$

is used to represent a process which indefinitely repeats the action a. One can have a combination of choice and recursion to encode a process which goes on engaging in a sequence of actions until a certain action happens, when it can change its behavior. For example,

$$P = a \rightarrow Q \mid b \rightarrow P$$

represents a process that can either engage in an indefinitely long sequence of action b, or if it can engage in action a, then it changes its behavior according to process described by the term Q.

Given that we are now able to represent sequential processes with the above described notations, we can describe *parallel composition* with *rendez-vous* synchronizations. The parallel operator is denoted by $||$, and the synchronization alphabet accompanying the parallel operator notation. However, to avoid complicated notation, we will specifically mention the synchronization actions separately rather than burdening

the parallel composition operator. Let us assume that process P engages in actions in the set $\alpha(P) = \{a, b, c\}$, and process Q engages in actions in the set $\alpha(Q) = \{a, b, d, e\}$, then the parallelly composed process $P \parallel Q$ must synchronize on the action set $\alpha(P) \cap \alpha(Q) = \{a, b\}$. This means that whenever process P will come to a point when it is ready to engage in action a, process Q must be ready to engage in action a, and they would execute action a together. If Q is not ready for engaging in action a, at the time P is ready to do so, P must get suspended until Q executes further and gets ready to engage in this synchronization action. This is what is known as *rendez-vous* synchronization. For example,

$$P = a \rightarrow c \rightarrow b \rightarrow P,$$

and

$$Q = d \rightarrow a \rightarrow e \rightarrow d \rightarrow b \rightarrow Q,$$

when parallely composed into $P \parallel Q$, then P has to remain suspended until Q finishes its independent action d, and then they can execute the synchronous action a together, after which P can do its independent action c while Q does its sequence of e and d, before the two of them synchronize again on b.

Given this background, we are ready to describe the *Dining Philosopher* solution based on [28], which are attributed to E. W. Dijkstra and C. Scholten.

As per the figure shown, we have 5 philosopher CSP threads, and 5 fork CSP threads. The philosopher processes are called $PHIL_i$ where $i = 0..4$, and the fork processes are denoted as $FORK_i$ where $i = 0..4$. Due to the symmetric nature of the processes, we will describe the ith process only. The action alphabet of $PHIL_i$ is given as

$$\alpha(PHIL_i) = \{think_i, requestseat_i, getfork^i_{i-1}, getfork^i_{i+1}, eat_i, \\ dropfork^i_{i-1}, dropfork^i_{i+1}, relinquishseat_i\}.$$

Similarly for the fork processes

$$\alpha(FORK_j) = \{getfork^{j-1}_j, getfork^{j+1}_j, dropfork^{j-1}_j, \\ dropfork^{j+1}_j\}.$$

Note that $i-1, j-1, i+1, j-1$ are modulo 5 increment and decrement operations. Which means when $i = 4, i + 1 = 0$, similarly, when $i = 0, i - 1 = 4$ etc. Now we can write down the CSP process describing the behavior of the philosopher number i as follows:

$$PHIL_i = think_i \rightarrow requestseat_i \rightarrow getfork^i_{i-1} \rightarrow getfork^i_{i+1} \rightarrow eat_i \\ \rightarrow dropfork^i_{i-1} \rightarrow dropfork^i_{i+1} \rightarrow relinquishseat_i \rightarrow PHIL_i.$$

Similarly, behavior of the process describing the fork number j is given as

$$FORK_j = getfork_j^{j-1} \rightarrow dropfork_j^{j-1} \rightarrow FORK_j \overset{\rightarrow}{|} getfork_j^{j+1} \rightarrow dropfork_j^{j+1} \rightarrow FORK_j$$

Note that in this particular solution, $FORK_j$ prioritizes between the philosopher at its left and the one at its right, so that the solution is tenable to the case when the number of philosophers is even as well.

So the dining philosopher system can now be written as

$$DP = ||_{i=0}^{i=4} (PHIL_i \,||\, FORK_i)$$

where $||_{i=0}^{i=4}$ is an obvious indexed form of the parallel composition operator.

Unfortunately as explained in [28], this solution can deadlock when all the philosophers decide to ask for the fork on their left at the same time. This necessitates an arbitrator, and a solution is given in the form of a footman, invented by C. S. Scholten. In our examples, we use this footman based solution to illustrate a finite state machine control of the dining philosopher system. Here we describe in CSP notation, how the footman works. Basically, the footman is the one who receives request for a seat by a hungry philosopher, and if the number of philosophers eating at that time is less than four, only then it grants such a request. This avoids the deadlock scenario described above. We describe the CSP notational form of the footman following [28] with a mutually recursive definition. Let us denote the footman as $FOOT_n^S$ for $n = 0..4$, such that $FOOT_n^S$ denotes the behavior of the footman when n philosophers are seated and the set S contains the indices of those who are seated. The alphabet of $FOOT_n^S$ is given by $\cup_{i=0}^{i=4}\{requestseat_i, relinquishseat_i\}$, and hence those are the actions of $PHIL_i$ which needs to synchronize with the footman. Now we describe the CSP terms for the $FOOT_n^S$ as follows:

$$\begin{aligned}
FOOT_0^{\{\}} &= |_{i=0}^{i=4} (requestseat_i \rightarrow FOOT_1^{\{i\}}) \\
FOOT_1^{\{i\}} &= (|_{j \neq i} (requestseat_j \rightarrow FOOT_2^{\{i,j\}})) \\
&\quad | (relinquish_i \rightarrow FOOT_0) \\
FOOT_2^{\{i,j\}} &= (|_{k \neq i,j} (requestseat_k \rightarrow FOOT_3^{\{i,j,k\}})) \\
&\quad | (|_{l \in \{i,j\}} (relinquish_l \rightarrow FOOT_1^{\{i,j\}-\{l\}})) \\
FOOT_3^{\{i,j,k\}} &= (|_{l \neq i,j,k} (requestseat_l \rightarrow FOOT_4^{\{i,j,k,l\}})) \\
&\quad | (|_{x \in \{i,j,k\}} (relinquish_x \rightarrow FOOT_2^{\{i,j,k\}-\{x\}}))
\end{aligned}$$

$$FOOT_4^{\{i,j,k,l\}} =|_{x\in\{i,j,k,l\}} (relinquish_x \rightarrow FOOT_3^{\{i,j,k,l\}-\{x\}})$$

So with the footman the dining philosopher system is now described as

$$DP = FOOT_0^{\{\}} \;||\; (||_{i=0}^{i=4} (PHIL_i \;||\; FORK_i))$$

We will show in later chapters that each $PHIL_i$ can contain computations which are more involved rather than being just atomic actions as shown here. In fact, the computations involved may actually require computation in another MoC. The footman can be also implemented in FSM MoC, rather than keeping it a CSP process. Such examples are provided in this book to create a running example of multi-MoC modeling while keeping it simple.

5.3 Finite State Machine in SystemC

The Finite State Machine (FSM) model in SystemC is inherent to the modeling constructs of SystemC. The existing SystemC can effectively construct FSM models. Some designers may argue that given a Discrete-Event simulation kernel, there is no need to add a Finite State Machine (FSM) kernel for SystemC. However, with the vision of a truly heterogeneous and hierarchical modeling environment in SystemC, the need for such an inclusion is important. We categorize the FSM Model of Computation as a specialized Discrete-Event (DE) Model of Computation that provides the designer additional modeling fidelity. A simple example of an FSM is an even/odd parity checker as shown in Figure 2.6.

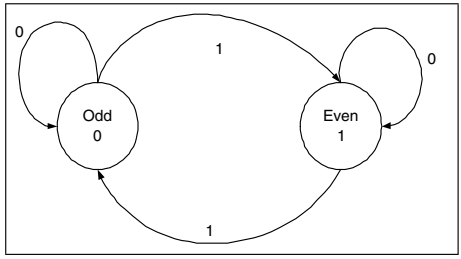


Figure 2.6. FSM Implementation of a Parity Checker

A parity checker receives a sequence of bits as inputs and if the number of 1's received is even then the output is 1, otherwise 0. The state transition table is shown in Table 2.1.

Based on the initial state and the input bit, the next state of the FSM is determined. Pictorially, this can be represented in a graph form

Table 2.1. State Table for Parity Checker

Present State	Input	Next State/Output
0	0	0
0	1	1
1	0	1
1	1	0

shown in Figure 2.6. The transitions from even to odd only occur when the input is 1, but if the input is 0 then the FSM remains in its same state satisfying the parity bit checker. We elaborate our implementation of the FSM kernel in SystemC in Chapter 7.

Note that formally an FSM could be either a Mealy style FSM or a Moore style FSM. In the example of the parity checker, output is determined by the current state of the machine, and hence it is a Moore style FSM. We often need Mealy style FSMs as well. In the current version of the FSM kernel we have not made any distinction of the two, and our FSM structure can support both.

We end this section with the example of the footman discussed in the previous section as a Moore style FSM.

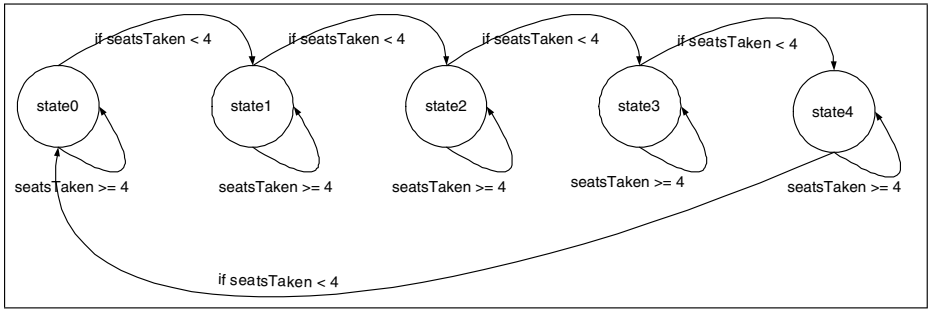


Figure 2.7. FSM implementation of the Footman

Figure 2.7 shows five states where the initial state is *state0*. A *requestseat* from a philosopher results in the FSM taking a transition to *state1* and allowing the philosopher to seat at the table. This continues through *state2* and *state3*. Once the FSM takes the transition to *state4*, four seats have already been taken and no more philosophers will sit to eat. Hence, all other *requestseat* will result in returning back to the same *state4*. After one of the philosophers is done eating, the FSM takes the transition back to *state0*. For details on the workings of the footman, we refer to the reader to the earlier section discussing the CSP Dining Philosophers example.

Chapter 3

SYSTEMC DISCRETE-EVENT KERNEL

1. DE Simulation Semantics

Modification of the existing SystemC Discrete-Event (DE) kernel requires study of the reference implementation source code. This section provides an implementation overview of the Discrete-Event kernel in SystemC. We provide code fragments as well as pseudo-code to identify some aspects of the source code with the assumption that the reader has basic application-level design knowledge of SystemC. For details on the QuickThread [35] implementation in SystemC refer to Appendix A.

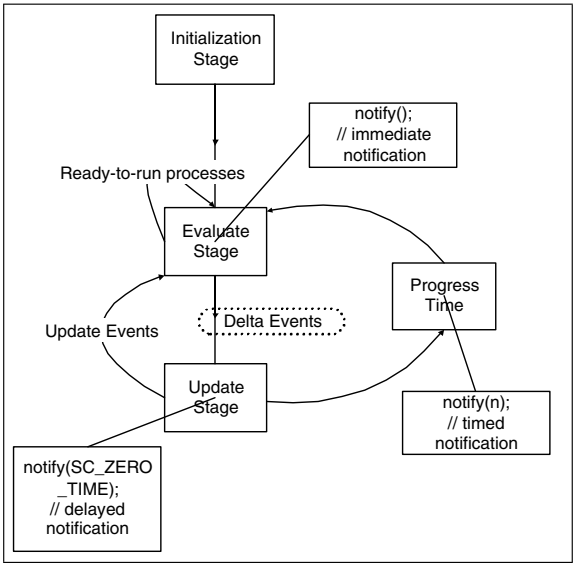


Figure 3.1. Discrete-Event Kernel Simulation Semantics

Figure 1 shows a pictorial representation of SystemC's simulation kernel. Current implementation of the DE kernel in SystemC follows the *Evaluate-Update* paradigm to simulate a model. Before introducing simulation semantics of the SystemC DE kernel, we familiarize the reader with some terminology. There are three types of SystemC processes: *SC_METHOD()*, *SC_THREAD()* and *SC_CTHREAD()*. When any of these processes are *ready-to-run*, they are pushed onto their respective *runnable* queues. Processes become *ready-to-run* when events are generated on the signals/channels sensitive to the particular process, with the exception of processes during initialization. During initialization all processes are made *ready-to-run* unless specifically specified with *dont_initialize()*. Events on signals/channels call the *request_update()* function that notifies the kernel that an update is required on the signal or channel. The Update phase invokes the *update()* function on all pending events.

The semantics of the SystemC kernel as described in the SystemC Language Reference Manual (LRM) [49] contains the following:

1 **Initialization**
Initialization is the first stage the SystemC kernel enters upon starting the simulation. In this stage, every SystemC process executes once unless the process is specified with a *dont_initialize()*. This prevents the kernel from making the process *ready-to-run* during initialization. *SC_METHOD()* processes execute once until completion, and *SC_THREAD()* processes execute until a *wait(...)* statement is reached at which the respective process suspends itself. The order of execution for these processes is unspecified.

2 **Evaluate Phase**
This stage executes processes that are *ready-to-run*, until there are no runnable processes on the *runnable* queues. If the process is an *SC_THREAD()* or *SC_CTHREAD()*, then these processes execute until a *wait(...)* or *wait_until(...)* are encountered, and if the process is an *SC_METHOD()*, then it is executed until completion. Once again, the order in which the processes are selected is unspecified in the standard. Immediate event notifications may occur from the execution of a process causing additional processes to be *ready-to-run* during the same Evaluate phase. During the execution of these processes, the signals/channels may invoke a *request_update()* that schedules the signals/channels for pending calls to the *update()* function during the Update phase.

3 **Repeat of Evaluate Phase for *ready-to-run* processes**
The processes that become *ready-to-run* during the Evaluate phase

require being evaluated, looping back through the Evaluate phase. Hence, referring back to stage 2 for any other process that is *ready-to-run*.

4 **Update Phase**
This stage services all pending calls scheduled during the Evaluate phase by the *request_update()* function. The *update()* function is called on the signals/channels requesting an update. This *update()* may cause other processes to become *ready-to-run*.

5 **Check delta-delayed notifications**
If there are any pending delta-delayed notifications, determine which processes are to be executed and return to stage 2.

6 **Check timed notifications**
Check if there are any remaining timed event notifications. If there are none, then the simulation has completed.

7 **Advance of Simulation time**
Otherwise, advance the simulation time to the earliest next pending timed event notification.

8 **Iterate**
The processes that become *ready-to-run* due to the pending notifications at the current time, have to be identified and processed through the Evaluate phase by returning to stage 2.

1.1 Delta Cycles and Delta Delays

From the simulation semantics, it is clear that the notion of *delta cycles* and *delta delays* are used in SystemC. Delta-cycles are small steps in simulation time that do not advance the simulation time, commonly used for interpreting zero-delay semantics. Processing a delta cycle executes actions scheduled at the current simulation time using the Evaluate-Update paradigm without advancing the simulation time. Delta-delays schedule a process to execute in the following delta cycles, but once again the simulation time is not advanced. The notion of delta cycles is essential in performing simultaneous actions and in reality provides a partial order for these actions.

2. Implementation Specifics

The original DE kernel is a queue-based implementation where the primary data-structure used is a queue. Most of the code fragments are extracted from the *sc_simcontext.cpp* file present in the kernel source directory.

The simulation starts once the *sc_start(...)* function is called from the modeler's program code. This executes the kernel by proceeding through the initialization phase where several flags are set and coroutine packages are prepared for use. Then, all *SC_METHOD()* and *SC_THREAD()* processes are made runnable. The *runnable* queues and process lists are merely lists of pointer handles. The *SC_METHOD()*, *SC_THREAD()*, and *SC_CTHREAD()* vector lists display the lists used to store pointers to the processes in Listing 3.1. The
m_delta_events is the list that holds the address of the events generated as delta events. The type for the lists used in the kernel are typedefed as shown in Listing 3.1.

Listing 3.1. Process vector lists

```
1 typedef sc_pvector<sc_method_handle>  sc_method_vec;
2 typedef sc_pvector<sc_thread_handle>  sc_thread_vec;
3 typedef sc_pvector<sc_cthread_handle> sc_cthread_vec;
4
5 sc_pvector<sc_event*> m_delta_events;
```

Using these typedef types, three lists are instantiated for the corresponding *SC_METHOD()*, *SC_THREAD()*, and *SC_CTHREAD()* processes as private members of the class *sc_process_table*, whereas
*m_delta_event*s is simply an *sc_pvector<...>* list.

We first present the pseudo-code for the simulation kernel followed by explanation of the functions and structures present. The pseudo-code is shown in Listing 3.2.

3. Discrete-Event Simulation Kernel

There is pseudo-code made available in the SystemC Functional Specifications for further reference [49]. However, the pseudo-code in Listing 3.2 depicts the program outline for the DE kernel using the Evaluate-Update paradigm as interpreted from the source code. The pseudo-code is marked with the stages that correspond to the above mentioned Evaluate-Update paradigm using the C++ comment format to provide better separation for each of the stages.

The first step during initialization is initializing the flags and variables that are used in the simulation, after which the QuickThread packages are prepared [35]. This is followed by making all *SC_METHOD()*s, *SC_THREAD()*s and *SC_CTHREAD()*s *ready-to-run* such that the simulation can begin executing them [Listing 3.2, Line 10 - 11]. The *crunch()* function is called once from within the initialization stage to process the Evaluate-Update stages [Listing 3.2, Line 31]. The order in which the

Listing 3.2. Discrete Event Kernel

```
void sc_simcontext::initialize( bool no_crunch ) {

  // stage 1
  // begin UPDATE phase
  perform update on primitive channel registries;

  initialize thread package;
  prepare all thread and cthread processes for simulation;

  make all method process runnable;
  make all thread process runnable;

  process delta notifications;
  // run the DELTA cycles
  run crunch();
  }

void sc_simcontext::simulate( const sc_time& duration ) {
  // stage 1
  initialize( true );
  do {
    do {
      run crunch();
      // stage 6 and 7
      process timed notifications;
       } until no timed notifications or runnable processes;
  // stage 8
  } until duration specified for simulation
}

void sc_simcontext::crunch() {

  // stage 2
  // EVALUATE phase
  while( true ) {
    execute all method processes in method_vec list;
    execute all c/thread processes in thread_vec list;

    //stage 3
    break when no more processes to run;
  }
  // stage 4
  // UPDATE PHASE
  update primitive channel registries;

  // stage 5
  process delta notifications;
} // End
```

processes are executed is unspecified to model the nature of real hardware. The processes can be withheld from being executed during the initialization stage if a *dont_initialize()* function call is invoked in the constructor of the object. This will cause the kernel to not schedule that process for execution during initialization.

The *crunch()* function first begins with the Evaluate phase where all *SC_METHOD()*, *SC_THREAD()* and *SC_CTHREAD()* processes are executed with respect to their sensitivity lists. Execution of processes

can ready other processes to execute, *request_update()* on signals, channels and events signifying that those entities require updates. Once there are no more processes to execute, the Update phase begins. This is where the *perform_update()* function is called to perform the updates. All delta events generated are stored in a list called *m_delta_events*, which is a list of pointers to events of type *sc_event*. These delta events are then all processed that in-turn result to pending notifications. This might ready more processes to execute which causes the repetition of the Evaluate-Update paradigm until the end of simulation. The end of simulation is realized when there are no timed notifications remaining or an *sc_stop()* is encountered, but if there are pending timed notifications, then the scheduler advances the current simulation time to the earliest pending timed notification and continues to iterate through the Evaluate-Update phases until all timed notifications have been processed.

The *simulate(...)* function [Listing 3.2, Line 18] calls the initialization function and the *crunch(...)* function for the execution of Evaluate-Update on processes for the duration of the simulation. The *sc_start(...)* function call to begin the SystemC simulation calls this *simulate(...)* function. This overloaded function *sc_start(...)* can take in the parameter for the duration the simulation is to execute. This value is passed onto the kernel as the variable. *initialize(...)* readies the simulation for execution and processes the *crunch()* once. This is where the processes are executed once during initialization.

The *crunch(...)* function is responsible for executing the processes through the Evaluate-Update paradigm. It is repeatedly called from *simulate()* for the duration of the simulation or if an *sc_stop()* statement is invoked from the modeler's code. The *sc_stop()* function terminates the simulation at that instance. The Evaluate phase in the *crunch()* function performs the task of executing the runnable *SC_METHOD()* processes first, after which the *SC_THREAD()*s and *SC_CTHREAD()*s are executed. The *runnable* queues are popped to retrieve the process handles through which the member function *execute()* is invoked. Process handles are pushed onto the *runnable* queues through events on the primitive channels. A check for *sc_stop()* is made to ensure that the programmer has not invoked the termination of simulation. This concludes the Evaluate phase and begins the Update phase where the primitive channels are updated using the *perform_update()* function call. This is when all the signals and events that called the *request_update()* function are updated. The signals, channels and event values are updated according to the designer specification. For events, this is an important aspect to understand because there can be immediate, *SC_ZERO_TIME*, and timed notifications on events. Their respective calls are *notify()*, *no-*

tify(SC_ZERO_TIME) and *notify(n, sc_time_unit)* where n can be an integer and *sc_time_unit* is simply a unit of time such as *SC_NS*. Immediate notification results in the event being updated immediately in the same delta cycle, *SC_ZERO_TIME* notification means the event is updated in the next delta cycle, and timed notifications are updated after n units of time. Accordingly, the delta events are processed until no delta events are left to be processed causing the simulation time to advance to the next pending timed notification and repeating this until the end of simulation.

4. Example: DE Kernel

In effort to clarify the Discrete-Event kernel, we provide an example that exercises some of the properties such as immediate and delayed notification as well as the scheduling of all processes.

Listing 3.3 displays the primary entry functions and *SC_METHOD()* and *SC_THREAD()* definitions. We provide a brief description of the processes in this example and how they should function according to the simple program.

P1_meth is an *SC_METHOD()* process that updates the local variable called *temp* with the value read from signal *s2* [Listing 3.3, Line 23]. This value is incremented and written to signal *s1* after which an event is scheduled on event *e2* after 2 nanoseconds (ns). The signal *s1* gets updated during the current delta cycle's Update phase. The value is ready on that signal at the end of that delta cycle. That completes the execution of the *SC_METHOD()*. However, *P1_meth* will only fire when there is a change in signal *s2* due to the sensitivity list definition. This process is declared as *dont_initialize()*, defining that this process should not be executed during initialization phase.

The remaining processes are all of *SC_THREAD()* types. *P4_thrd* and *P5_thrd* are straightforward in that they wait on events *e1* and *e2* respectively [Listing 3.3, Line 46 & 53]. Upon receiving an event the remaining of the code segment is executed until the *wait(...)* statement is encountered again. At a wait statement, the executing process suspends allowing other processes to execute. The sole purpose of these two *SC_THREAD()*s is to print to the screen when the event is notified along with helper functions in identifying the changes on signals. *P3_thrd* is similar in that it waits for events except in this case it waits for either event to be notified.

P2_thrd has an internal counter called *local_counter* that initializes the *s2* signal with -1 during the initialization stage and increments from there on to write to the same signal [Listing 3.3, Line 29]. After the increment is done, an immediate notification is sent on event *e1*. This

Listing 3.3. Discrete-Event Simulation Example

```
SC_MODULE(DE_KERNEL)  {
  sc_in<bool> ck; sc_out<int> s1;  // ports
  sc_signal<int> s2; //signals
  sc_event e1, e2, // events
  void P1_meth(); void P3_thrd(); void P2_thrd();
  void P4_thrd(); void P5_thrd(); // functions

 SC_CTOR(DE_KERNEL) {
  SC_THREAD(P5_thrd);
  SC_THREAD(P4_thrd);
  SC_THREAD(P3_thrd);
    sensitive << ck.pos();
  SC_THREAD(P2_thrd);
    sensitive << ck.pos();
  SC_METHOD(P1_meth);
     dont_initialize();
    sensitive << s2;
}/* END DE_KERNEL C_TOR*/

  private:
   int temp;

  void DE_KERNEL::P1_meth() {
       temp = s2.read();
       s1.write(temp+1);
       e2.notify(2,SC_NS);
  }/* END P1_meth */

  void DE_KERNEL::P2_thrd() {
       int local_counter = -1;
       while (true)  {
            s2.write(local_counter);
           ++local_counter;
            e1.notify();//immediate
            wait();
       }/* END WHILE */
  }/* END P2_thrd */

  void DE_KERNEL::P3_thrd() {
       while(true) {
            wait(e1|e2);
            cout << "sc_time_stamp :: \t" << sc_time_stamp() <<
                endl;
       }/* END WHILE */
  }/* END P3_thrd */

void DE_KERNEL::P4_thrd() {
  while (true)  {
    wait(e1);
    cout << sc_time_stamp() << " \t event 1 triggered" << endl;
    }/* END WHILE */
}/* END P4_thrd */

void DE_KERNEL::P5_thrd() {
  while(true) {
       wait(e2);
       cout << sc_time_stamp() << " \t event 2 triggered" << endl
           ;
  }/* END WHILE */
}/* END P5_thrd */
```

indicates processes that are waiting on *wait(e1)* to resume and continue execution. We do not present *wait(SC_ZERO_TIME)* but that can be understood as a notification in the following delta cycle instead of the same one.

This simple program inspired by [2] illustrates the basic workings of the SystemC DE kernel with respect to events, signals, sensitivities and processes, while introducing some of the basic functions of the HDL. To prepare for the following chapters, we provide another example that will be carried onto the next chapter for further explanation. We choose the FIR model from the SystemC distribution [49].

4.1 Finite Impulse Response Example

Listing 3.4. stimulus.h

```
1SC_MODULE(stimulus) {
2
3   sc_out<bool> reset;
4   sc_out<bool> input_valid;
5   sc_out<int>  sample;
6   sc_in<bool>  CLK;
7
8   sc_int<8>      send_value1;
9   unsigned cycle;
10
11  SC_CTOR(stimulus)
12  {
13      SC_METHOD(entry);
14      dont_initialize();
15      sensitive_pos(CLK);
16      send_value1 = 0;
17      cycle       = 0;
18  }
19  void entry();
20};
```

This example is a three-staged module example at a higher level of abstraction than an RTL model. The FIR model is separated into three functional blocks as shown in Figure 3.2. The input block generates inputs for the FIR, the FIR block performs the Finite Impulse Response and the output block simply sends it to the standard output.

Figure 3.2. Discrete-Event FIR Block Diagram

Listing 3.5. stimulus.cpp

```
1 void stimulus::entry() {
2
3   cycle++;
4   // sending some reset values
5   if (cycle<4) {
6     reset.write(true);
7     input_valid.write(false);
8   } else {
9     reset.write(false);
10    input_valid.write( false );
11    // sending normal mode values
12    if (cycle%10==0) {
13      input_valid.write(true);
14      sample.write( (int)send_value1 );
15      send_value1++;
16    };
17  }
18 }
```

Listing 3.6. fir.h

```
1 SC_MODULE(fir) {
2
3   sc_in<bool>  reset;
4   sc_in<bool>  input_valid;
5   sc_in<int>   sample;
6   sc_out<bool> output_data_ready;
7   sc_out<int>  result;
8   sc_in_clk    CLK;
9
10  sc_int<9> coefs[16];
11
12  SC_CTOR(fir)
13      {
14       SC_CTHREAD(entry, CLK.pos());
15       watching(reset.delayed() == true);
16       #include "fir_const.h"
17      }
18
19  void entry();
20 };
```

Here we describe the DE code and in Chapter 5 we compare it with the corresponding SDF FIR model. Listing 3.4 and 3.5 present the Stimulus block from Figure 3.2, where both the *SC_MODULE()* declaration and the *entry()* function are displayed. The Stimulus block as can be seen from Listing 3.5 has a reset state for four cycles after which every ten cycles an incremented *send_value1* is sent out on the sample port [Listing 3.5, Line 5 - 15]. Note that the Stimulus block has four ports, clock input port, output port for the *sample*, *input_valid* output port to indicate to the next block (FIR block) that a sample has been generated, and an output *reset* port to perform reset on all the modules. Also note that this

functional block is an *SC_METHOD()* process sensitive to the positive edge of the clock [Listing 3.4, Line 3 - 6].

Similarly, the FIR block shows input ports accepting the *sample* and *input_valid* signals from the Stimulus block [Listing 3.6, Line 5 & 4]. A clock input port is instantiated because this is an *SC_CTHREAD()* process. In addition, this FIR block needs to have control signals sent to the Display block and therefore has the *output_data_ready* output port to signal the Display block that the computed FIR value is present on the *result* output signal [Listing 3.6, Line 6 & 7].

The Display block simply has input ports that accept the data sent from the FIR block and the control signal allowing the Display block

Listing 3.7. fir.cpp

```
1#include <systemc.h>
2#include "fir.h"
3
4void fir::entry() {
5
6  sc_int<8> sample_tmp;
7  sc_int<17> pro;
8  sc_int<19> acc;
9  sc_int<8> shift[16];
10
11 // reset watching
12 /* this would be an unrolled loop */
13 for (int i=0; i<=15; i++)
14   shift[i] = 0;
15 result.write(0);
16 output_data_ready.write(false);
17 wait();
18
19 // main functionality
20 while(1) {
21   output_data_ready.write(false);
22   wait_until(input_valid.delayed() == true);
23   sample_tmp = sample.read();
24   acc = sample_tmp*coefs[0];
25
26   for(int i=14; i>=0; i--) {
27     /* this would be an unrolled loop */
28     pro = shift[i]*coefs[i+1];
29     acc += pro;
30   };
31
32   for(int i=14; i>=0; i--) {
33     /* this would be an unrolled loop */
34     shift[i+1] = shift[i];
35   };
36
37   shift[0] = sample_tmp;
38   // write output values
39   result.write((int)acc);
40   output_data_ready.write(true);
41   wait();
42 };
43}
```

Listing 3.8. display.h

```
1 SC_MODULE(display) {
2
3   sc_in<bool>  output_data_ready;
4   sc_in<int>   result;
5
6   int i, tmp1;
7
8   SC_CTOR(display)
9     {
10      SC_METHOD(entry);
11      dont_initialize();
12      sensitive_pos(output_data_ready);
13      i = 0;
14    }
15
16  void entry();
17 };
```

to execute. This is an *SC_METHOD()* process that is sensitive to the *output_data_ready* signal that it accepts from FIR [Listing 3.8, Line 3]. Note that we have inserted a counter i in Display that we use to terminate the simulation via *sc_stop()* once a certain number of outputs are computed. We could have simply inserted the number of cycles in the *sc_start(n)* function call but decided against it in order to maintain consistency between DE FIR and Synchronous Data Flow FIR models. Listing 3.10 describes the instantiation of the modules and their connections via signals.

The DE kernel begins preparing each module for execution during initialization. Notice that the Stimulus and Display block declaration have a *dont_initialize()* which makes the kernel realize that these modules are to be skipped for execution during initialization. The FIR block will be fired during initialization, but the *watching(...)* [Listing 3.6, Line

Listing 3.9. display.cpp

```
1 #include <systemc.h>
2 #include "display.h"
3
4 void display::entry(){
5
6   //  Reading Data when valid if high
7   tmp1 = result.read();
8   cout << tmp1 << endl;
9   i++;
10  if(i == 5000000) {
11    sc_stop();
12  };
13 }
14 // EOF
```

Listing 3.10. FIR main.cpp

```
#include <systemc.h>
#include "stimulus.h"
#include "display.h"
#include "fir.h"
#include "sdf_includes.h"
#include "sdf_structure.h"
int sc_main (int argc , char *argv[]) {
  sc_clock          clock;
  sc_signal<bool>   reset;
  sc_signal<bool>   input_valid;
  sc_signal<int>    sample;
  sc_signal<bool>   output_data_ready;
  sc_signal<int>    result;

  stimulus stimulus1("stimulus_block");
  stimulus1.reset(reset);
  stimulus1.input_valid(input_valid);
  stimulus1.sample(sample);
  stimulus1.CLK(clock.signal());

  fir fir1( "process_body");
  fir1.reset(reset);
  fir1.input_valid(input_valid);
  fir1.sample(sample);
  fir1.output_data_ready(output_data_ready);
  fir1.result(result);
  fir1.CLK(clock);

  display  display1 ( "display");
  display1.output_data_ready(output_data_ready);
  display1.result(result);

  sc_start(clock, -1);
  return 0;
}
```

15] statement forces the module to only proceed execution once the condition inside *watching(...)* is satisfied, which in this case is *reset* being *true*. This mechanism ensures that the *reset* waits until four clock cycles are complete as shown in Listing 3.4 before the FIR block is executed. Dynamic scheduling is performed on this system with the help of control signals that direct the flow of tokens from the Stimulus to FIR and finally to the Display block.

Chapter 4

FEW WORDS ABOUT IMPLEMENTATION CLASS HIERARCHY

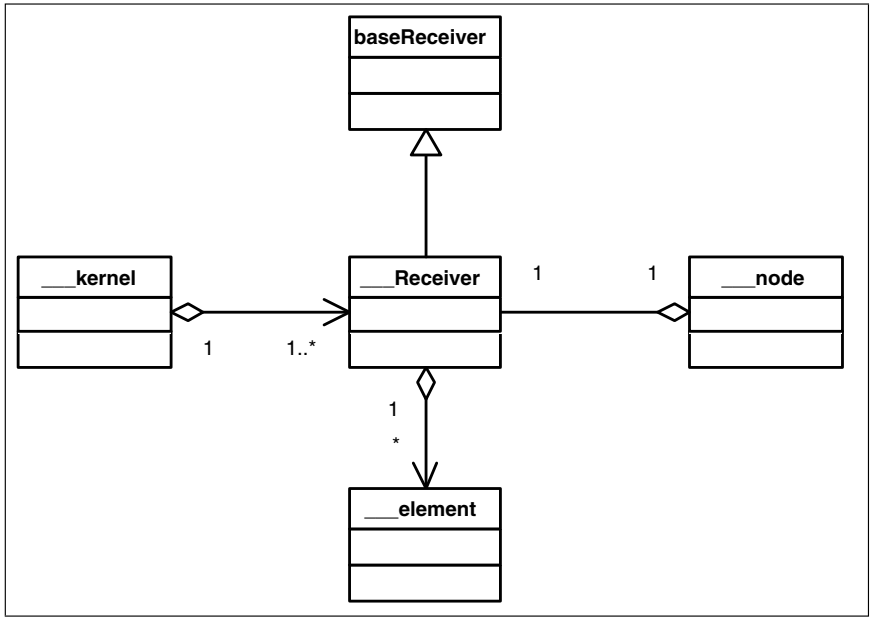


Figure 4.1. General Implementation Class Hierarchy

This chapter informs the reader about the organization of our kernel implementation. During the development process of alternative kernels for SystemC, several implementation hierarchies and data structures were investigated. In this book we did not make an effort to unify them. This book presents a snapshot of the current status of the project so that other interested developers can use the concepts and ideas to de-

velop their own multi-MoC kernels. Once the reader goes through the Synchronous Data Flow, Communicating Sequential Processes and Finite State Machine chapters, a distinct difference in class hierarchy with the SystemC kernel development can be noticed. We expect further gradual changes in implementation hierarchy in the future as we improve our SystemC kernel implementations. Nonetheless, we propose a hierarchy that allows for an extensible design for multi-MoC modeling. We simply lay a foundation that can support this, but do not currently employ it to its maximum potential.

In general, the class hierarchy resembles the class diagram shown in Figure 4.1. Some of the terminology used in Figure 4.1 are borrowed from [25]. It is not necessary to strictly conform to this class hierarchy, because some implementations do not require such a class structure and some require more encapsulation. The terminology used are as follows:

Kernel: A class that allows for creation and simulation of multiple instances of a model represented by a specific MoC.

Node: A representation of a specific function block that exhibits behavior specified by the MoC. For example, a *CSPnode* is a representation for a CSP process by encapsulating an instance of *CSPReceiver*.

Receiver: An encapsulation class to separate the data structure of an MoC from its communication with the designer and MoC-specific nodes. The common functionalities can be derived from a *baseReceiver* class.

baseReceiver: A class that encapsulates common functionalities and data structures employed by a receiver. Examples are queues that are used in DE and CSP MoCs as runnable lists and graph structures as used in representing an SDF and CSP model.

Element: Embedded deepest of all classes in terms of class hierarchy, an *element* class defines a structure that aids in creating the main data structure used to construct a model for an MoC.

This class hierarchy is only to provide minimal organization in developing the additional kernels and classes for encapsulation and functionality. Additional classes if required, should be added for better object oriented programming practices.

The CSP kernel class diagram in Figure 4.2 illustrates the CSP kernel implementation loosely employing the general implementation class hierarchy.

A CSP model is best represented as a graph. This graph is represented in *CSPReceiver* by a list-based data structure using Standard Template

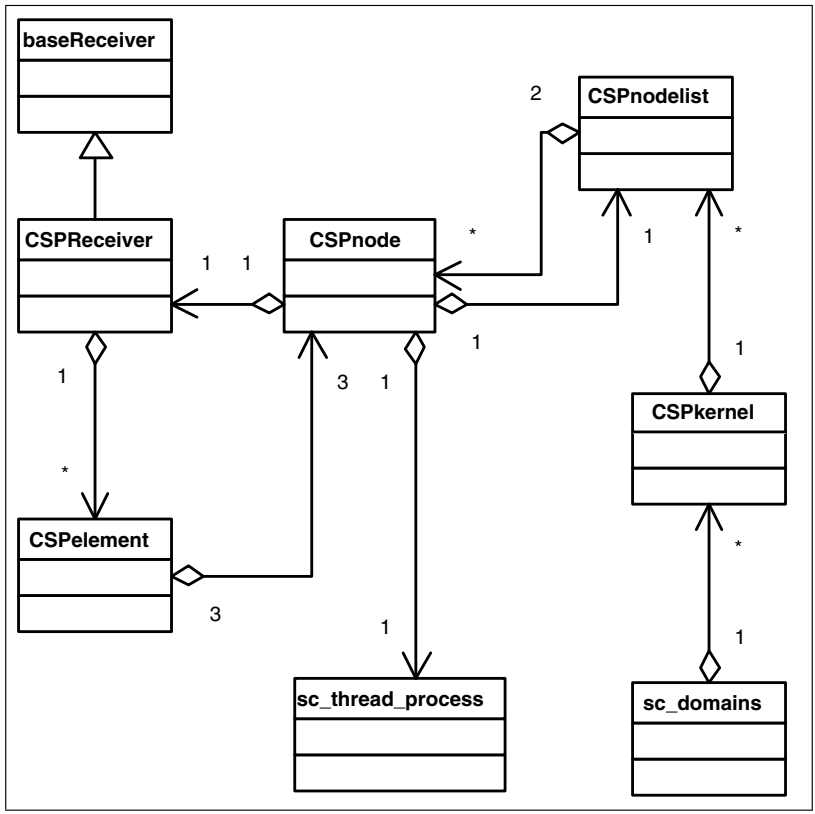


Figure 4.2. CSP Implementation Class Hierarchy

Library (STL) *vector* class. This list contains pointers to *CSPelement*. The *CSPelement* class provides the *CSPReceiver* with information about which nodes are connected via channels and the data to be transferred on each channel. *CSPchannel* inherits from *CSPelement* as shown later in this chapter, because in essence they exhibit the exact same behavior. A *CSPnode* has one instance of a *CSPReceiver* and contains a pointer to the SystemC thread class *sc_thread_process*. The modeler creates an instance of *CSPnode* within an *SC_MODULE(...)* to distinguish that module as a CSP module. An additional class called *CSPnodelist* holds pointers to *CSPnode*s and this class contains member functions that simulates the CSP MoC.

The FSM implementation hierarchy is simple where class *FSMReceiver* once again is a derived class from *baseReceiver*. However, the data structure present in this receiver uses a *map*<...> STL structure. This structure contains a *string* key field that holds the name of the state and a pointer to class *sc_method_handle*. *FSMnode* class is not

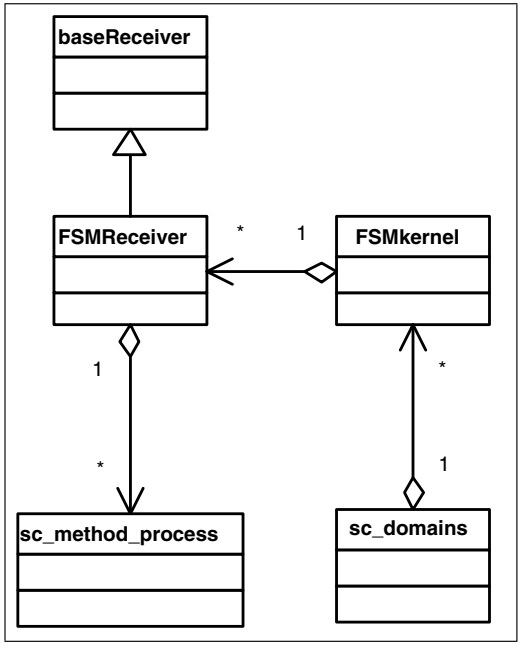


Figure 4.3. FSM Implementation Class Hierarchy

presented in Figure 4.3 because FSM transitions are explicitly defined within the state or entry function of that module. Hence, there is no need for communication between states other than signaling the FSM with the next state to carry out FSM simulation, which is done within the entry function of the process.

The *sc_domains* class in both Figure 4.2 and Figure 4.3 is a toplevel encapsulation class that implements the Application Protocol Interface (API). The purpose of *sc_domains* is to allow different kernels to interact with each other. Another use of the API is to allow for multiple models of the same MoC, for example three SDF models to function in the same model. Additional information regarding the API is discussed in Chapter 8.

1. MoC Specific Ports and Channels

MoC-specific ports and channels are needed due to the differences in communication protocols of the new kernels and the Discrete-Event kernel in SystemC. For example, SDF channels do not require the channels to generate events when pushing data onto a channel. Furthermore, the CSP rendez-vous semantics require its own event and event handling mechanism because the channels themselves play an important role in

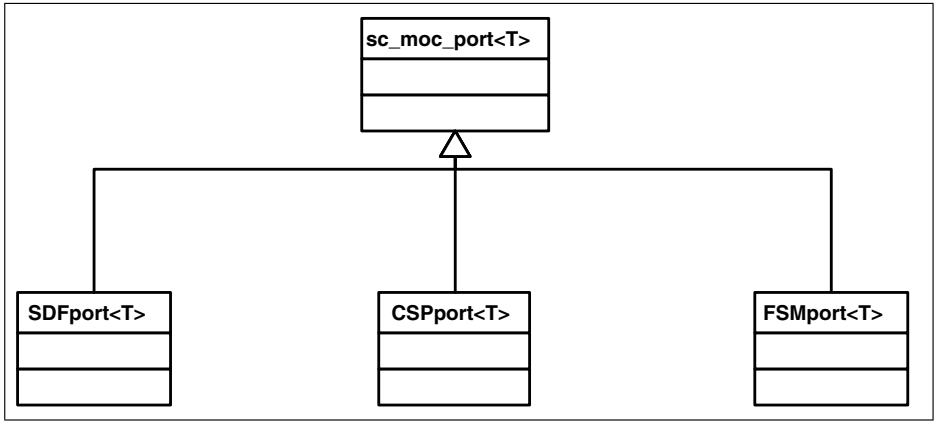


Figure 4.4. *sc_moc_port* Implementation Class Hierarchy

Table 4.1. Some Member functions of class *sc_moc_port*

Member Function	Purpose
operator()	Overloaded operator for binding a *sc_moc_channel* to a port.
getport()	Returns a pointer to the channel bound to this port.
push(...)	Inserts a value onto the channel
pop(...)	Returns the first value on the channel

the suspension and resumption of processes rather than simply providing a medium through which data is transferred.

MoC-Specific Ports

Figure 4.4 describes a class hierarchy for ports that accommodates multiple MoC communication. Basic functionality of the ports is implemented in *sc_moc_port* class and specializations are implemented in the derived classes. All the derived port classes are also polymorphic by making them *template* classes.

Listing 4.1 shows the class declaration and definition for *sc_moc_port*. The private data member of this class is a pointer to an *sc_moc_channel* object called *port*. This variable addresses the channel that is bound to this port. The roles of the member functions are shown in Table 4.1. Most of the generic roles of the port are implemented in the base class. If there is a need to add specific functionality for a particular port or channel then it can be added to the derived class.

Listing 4.1. class *sc_moc_port*

```
template <class T>  class sc_moc_port
{
  private:
    sc_moc_channel<T> * port;
    void bind(sc_moc_channel<T> * p);

  public:
    sc_moc_port<T>();
    ~sc_moc_port<T>();
    sc_moc_port(sc_moc_channel<T> * p);

    void operator () (sc_moc_channel<T> * port) { bind(port); };
    void operator () (sc_moc_channel<T> & port) { bind(&port);};
    T * getport();

    // Vector push/pop functions
    void push(T p);
    void push(T * p);
    T pop();

    void print()  { if (port != NULL) { port->print(); } };
};

template <class T >
  sc_moc_port<T>::sc_moc_port() {
    port = NULL;
    };

template <class T >
  sc_moc_port<T>::~sc_moc_port()  { };

template <class T >
  sc_moc_port<T>::sc_moc_port(sc_moc_channel<T> * p)  { port = p
      ; };

template < class T >
  void sc_moc_port<T>::bind(sc_moc_channel<T> * p)  { port = p
      ; };

template < class T>
  T * sc_moc_port<T>::getport() { return port; };

template < class T>
  void sc_moc_port<T>::push(T p)  { port->push(p);};
template < class T>
  void sc_moc_port<T>::push(T * p)  { port->push(p);};

template < class T>
  T sc_moc_port<T>::pop() { return(port->pop());  };
```

MoC-Specific Channels

Similarly, channels for the these MoC-specific ports follow a hierarchy displayed in Figure 4.5. The base class is *sc_moc_channel* from which the *SDFchannel*, *CSPchannel* and *FSMchannel* are all derived. Basic functions of a channel are implemented in the base class *sc_moc_channel* and MoC-specific channel implementations are contained in their respective derived class. The *SDFchannel* and *FSMchannel* are used to transport

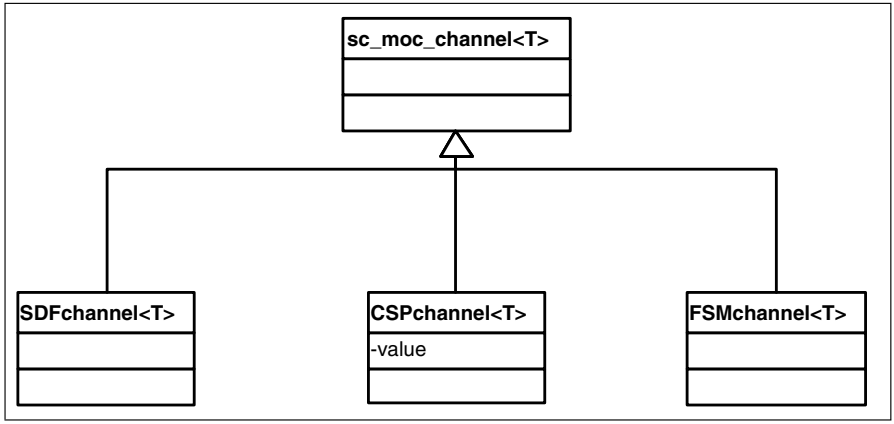


Figure 4.5. *sc_moc_channel* Implementation Class Hierarchy

data from one node to another. However, a *CSPchannel* plays an integral role in the rendez-vous communication, which requires additional implementation details to the channel. Figure 4.5 shows a data member *value* in *CSPchannel* that holds the value to be transferred once rendez-vous synchronization occurs. This *value* is of templated type allowing for all values of different types to be transferred. This is an example where specialization of an MoC-specific channel is done. Nonetheless, note that these MoC-specific channels and ports have no relation with *sc_channel* or *sc_port* and MoC-specific channels and ports are implemented using C++ data types.

Listing 4.2 displays the declaration and definition of the class *sc_moc_channel*. A list data structure is used to preserve the tokens pushed onto a channel. We use the *vector* class from STL. The *push(...)* member function inserts a value into the list and *pop()* returns the first value in the list. Note that both *sc_moc_port* and *sc_moc_channel* are template classes allowing for any type of data to be transferred through these ports and channels.

Note about SDF Ports and Channels

In Chapter 5, SDF ports and channels are not presented. However, the implementation does have SDF ports and SDF channels similar to FSM ports and FSM channels in Chapter 7. Class *SDFport* derives from *sc_moc_port* without any need for specialization since its purpose is to only transfer data and the *SDFchannel* is a derived class from *sc_moc_channel*. They are both templated classes as shown in Figure

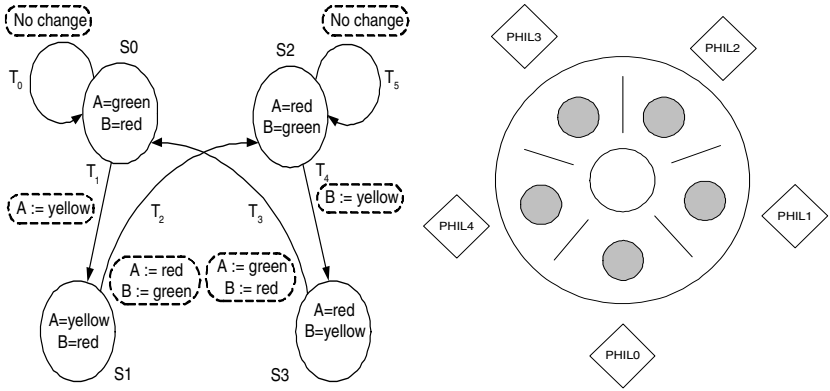
Listing 4.2. class *sc_moc_channel*

```
template <class T>
  class sc_moc_channel {
  private:
    vector<T> * mainQ;

  public:
    sc_moc_channel<T >(); // constructor
    ~sc_moc_channel<T>(); // destructor

    // Vector push/pop functions
    void push(T p);
    void push(T * p);
    T pop();

};

template <class T >
  sc_moc_channel<T>::sc_moc_channel() { mainQ = new vector<T>();
        };

template <class T >
  sc_moc_channel<T>::~sc_moc_channel()  { delete mainQ; };

template < class T>
  void sc_moc_channel<T>::push(T p) { mainQ->push_back(p);  };

template < class T>
  void sc_moc_channel<T>::push(T * p) { mainQ->push_back(*p); };

template < class T>
  T sc_moc_channel<T>:: pop() {
    T newT = mainQ->front();
    mainQ->erase(mainQ->begin());
    return (newT);
  }
```

4.4 and Figure 4.5. We provide SDF examples that employ the SDF ports and channels at our website [36].

baseReceiver class

baseReceiver currently only holds the receiver type, indicating whether an *FSMReceiver* or *CSPReceiver* has been derived. However, the usage of this base class can extend to encompass common data structures and helper functions. One such use of the *baseReceiver* can be to allow for implementation of the data structure required to represent MoCs that require a graph construction. We consider a graph like structure for SDF, FSM and CSP and currently we employ individual representations for each kernel shown in Figure 4.6. However, our current implementation does not unify the idea of representing commonly used data structures in the *baseReceiver* class and leave it aside as future work.



(a) FSM Traffic Light Controller (b) CSP Dining Philosopher

(c) SDF FIR

Figure 4.6. Graph-like Representation for SDF, FSM, CSP

2. Integration of Kernels

Kernel integration is a challenging task especially for kernels based on MoCs that exhibit different simulation semantics other than the existing DE scheduler semantics of SystemC. The MoC implementation chapters discuss the addition of a particular MoC in SystemC and documentation is provided describing the integration of these different MoCs. MoCs such as SDF and FSM are easy to integrate with the reference implementation and themselves, whereas CSP requires an understanding of QuickThreads [35] and the coroutine packages in SystemC. Integration of the SDF and FSM kernels are relatively straightforward, requiring minor additions to the existing source with the usage of Autoconf/Automake [21, 22]. In Appendix B we describe a method of adding newly created classes to the SystemC distribution using Autoconf/Automake. This approach is used for all MoC integration. However, the CSP kernel integration is non-trivial, which we describe in detail in Chapter 6.

Chapter 5

SYNCHRONOUS DATA FLOW KERNEL IN SYSTEMC

1. SDF MoC

This chapter describes our implementation of the Synchronous Data Flow (SDF) kernel in SystemC. We present code fragments for the SDF data structure, scheduling algorithms, kernel manipulations and designer guidelines for modeling using the SDF kernel along with an example.

The SDF MoC is a subset of the Data Flow paradigm [32]. This paradigm dictates that a program is divided into blocks and arcs, representing functionality and data paths, respectively. The program is represented as a directed graph connecting the function blocks with data arcs. From [53], Figure 5.1 shows an example of an SDF graph (SDFG). An SDF model imposes further constraints by defining the block to be invoked only when there is sufficient input samples available to carry out the computation by the function block, and blocks with no data input arcs can be scheduled for execution at any time.

In Figure 5.1, the numbers at the head and tail of the arcs represent the production rate of the block and consumption rate of the block respectively, and the numbers in the middle represent an identification number for the arc that we call arc labels. An invoked block consumes a fixed number of data samples on each input data arc and similarly expunges a fixed number of data samples on each of the output data arcs. Input and output data rates of each data arc for a block are known prior to the invocation of the block and behave as infinite FIFO queues. Please note that we interchangeably use function blocks, blocks and nodes for referring to blocks of code as symbolized in Figure 5.1 by A, B, C, D, E, F and G.

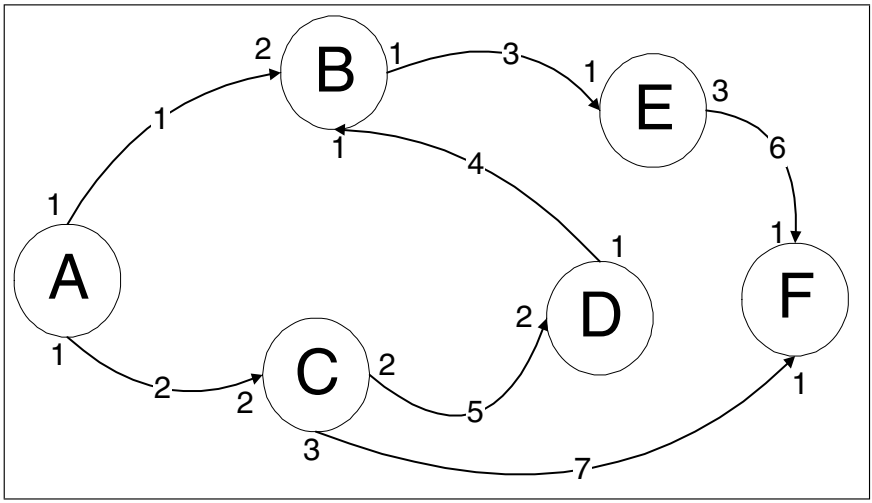


Figure 5.1. Example of a Synchronous Data Flow Graph [53].

Solution to static scheduling and execution of an *executable schedule* for SDF models in SystemC requires solutions to intermediary problems. The problems encountered are as follows:

1 Designing an appropriate data structure to contain information for an SDFG.

2 Constructing *executable schedule*s for the SDFGs. In this problem, there are two sub-problems. They are:

   (a) Computing the number of times each SDF block has to be fired that we refer to as the *repetition vector*.

   (b) Finding the order in which the SDF nodes are be executed, which we term *firing order*.

3 Designing a mechanism for heterogeneous execution of existing Discre te-Event (DE) and SDF kernel.

We define *repetition vector* and *firing order* based on [5]. By *repetition vector* we mean the number of times each function block in the SDFG is to be fired. However, a particular order is needed in which the function blocks in the SDFG are to be fired, that we refer to as the *firing order*. Constructing a *firing order* requires a valid *repetition vector*. A valid *executable schedule* refers to a correctly computed *repetition vector* and *firing order*. The directed nature of the graph and the production and consumption rates provide an algorithm with which the *firing order* is computed.

Problem 2a is discussed in [38] where Lee et al. describe a method whereby static scheduling of these function blocks is computed during compile time rather than runtime. We employ a modification of this technique in our SDF kernel for SystemC explained later in this chapter. The method utilizes the predefined consumption and production rates to construct a set of homogeneous system of linear Diophantine [11] equations. Solution to Problem 2b uses a scheduling algorithm from [5] that computes a *firing order* given that there exists a valid *repetition vector* and the SDFG is consistent. A consistent SDFG is a correctly constructed SDF model whose *executable schedule* can be computed.

We choose certain implementation guidelines to adhere to as closely as possible when implementing the SDF kernel.

1.1 General Implementation Guidelines

Implementation of the SDF kernel in SystemC is an addition to the existing classes of SystemC. Our efforts in isolating the SDF kernel from the existing SystemC kernel definitions introduces copying existing information into the SDF data structure. For example, the process name that is used to distinguish processes is accessible from the SDF data structure as well as existing SystemC process classes. The general guidelines we follow are:

Retain all SystemC version 2.0.1 functionality
: Current functionality that is provided with the stable release of SystemC 2.0.1 should be intact after the alterations related to the introduction of the SDF kernel.

SDF Data structure creation
: All SDF graph structure representation is performed internal to the kernel, hiding the information about the data structure, solver, scheduling algorithms from the designer.

Minimize designer access
: A separate SDF data structure is created to encapsulate the functionalities and behavior of the SDF. The modeler must only access this structure via member functions.

2. SDF Data Structure

Representing the SDF graph (SDFG) needs construction of a data structure to encapsulate information about the graph, such as the production and consumption rates, the manner in which the blocks are connected and so on. In this section, we describe the SDF data structure in detail with figures and code snippets. The majority of our imple-

mentation uses dynamically linked lists (*vector<...>*) provided in the Standard Template Library (STL) [13]. Figure 5.2 shows the SDF data structure. A toplevel list called *sdf_domain* of type *vector<sdf_graph*>* is instantiated that holds the address of every SDF model in the system. This allows multiple SDF models to interact with each other along with the DE models. Furthermore, each *sdf_graph* as shown in Listing 5.1 contains a vector list of pointers to *edges* which is the second *vector* shown in the Figure 5.2. Each *edge* object stores information about an SDF function block whose structure we present later in this section.

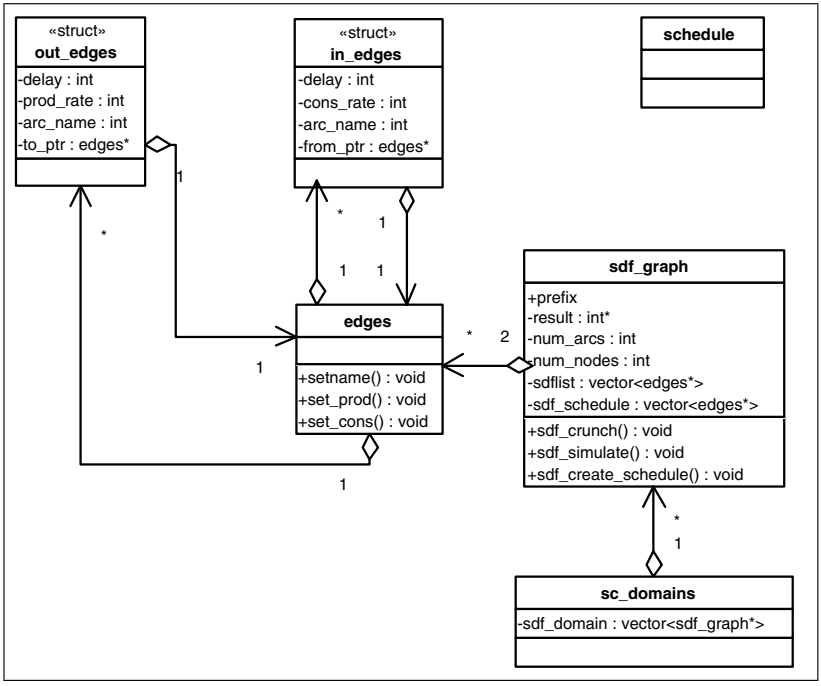


Figure 5.2. SDF Class Diagram

The toplevel class is defined as *sdf_graph* as shown in Listing 5.1. This is the class that holds information pertaining to a single Synchronous Data Flow Graph (SDFG). The SDFG encapsulates the following information: the number of blocks and arcs in the SDF, access to the *executable schedule* via *sdf_schedule*, the *repetition vector* through *result*, a string to identify the toplevel SDF by *prefix*, and a list of the SDF blocks represented by *sdflist* [Listing 5.1, Line 6 - 13].

All SDF blocks are inserted into the *vector<edges*> sdflist* list. This introduces the *edges* class that encapsulates the incoming and the outgoing arcs of that particular SDF block, an integer valued name for con-

Listing 5.1. class sdf_graph

```
class sdf_graph {
  public:
    sdf_graph();     // Constructor & destructor
    ~sdf_graph();

    vector< edges*> sdflist;     // SDF block list
    vector< edges*> sdf_schedule; // executable schedule
    int * result;

    int num_nodes;     // Number of blocks
    int num_arcs;    // Number of arcs

    string prefix;  // Store the name of the SDFG
};
```

structing the *repetition vector* and text based names for comparisons. We typedef this class to *SDFnode*.

Our implementation uses process names for comparison since the SystemC standard requires each object to contain a unique identifying name. When storing the process name either in *sdf_graph* class or *edges* class shown in Listing 5.3, we add a dot character followed by the name of the process.

Listing 5.2. Example showing name()

```
SC_MODULE(test) {

  // port declarations

  void entry1() {name();};
  void entry2() {name();};

  SC_CTOR(test) {
    name();
    SC_METHOD(entry1);  // first entry function
    SC_THREAD(entry2);  // second entry function
    // sensitivity list
  };
};
```

This is necessary to allow multiple process entry functions to be executed when the particular process is found. For clarification, Listing 5.2 presents an example showing SystemC's process naming convention. From Listing 5.2, it can be noticed that there are two entry functions *entry1()* and *entry2()*. Returning the *name()* function from within any of these functions concatenates the process name, and entry function name with a dot character in between. So, calling the *name()* function from *entry1()* will return "*test.entry1*"; calling the same from *entry2()* will return "*test.entry2*" and from the constructor will return "*test*".

Hence, a process name is a unique identifier describing the hierarchy of an entry function, for example "*test.entry1*". We require both these processes to execute for the process name "*test*" and to avoid much string parsing we use the substring matching function *strstr(...)*. However, this will also match a process name other than this module that might have an entry function with a name with "*test*". All processes with a prefix "*test.*" belong to the module "*test*". Therefore, the unique process name is constructed by adding the dot character after the process name and searching for that substring.

Listing 5.3. class edges

```
1 class edges {
2
3   public:
4     vector<out_edges> out;
5     vector<in_edges> in;
6
7     //constructor / destructor
8     edges();
9     ~edges();
10
11    // member functions
12    void set_name(string _name, vector<edges*> & _in); // set
          name
13    void set_prod(sc_method* to_ptr, int _prod); //set
          production rate
14    void set_cons(sc_method* from_ptr, int _cons); //set
          consumption rate
15
16    // variables that will remain public at the moment
17    string name;
18    int mapped_name;
19 };
```

The *edges* class encapsulates the incoming edges to an SDF block and outgoing edges from an SDF block. Lists *vector<out_edges> out* and *vector<in_edges> in* as shown in [Listing 5.3, Line 4 & 5] where *out_edges* and *in_edges* are of C type *struct*s as displayed in Listing 5.4 show the data structure used to store the outgoing arcs and incoming arcs respectively. Every *edge* object stores the process name as a string *name* and a corresponding integer value as *mapped_name* used in creating the topology matrices for the *repetition vector*s.

*struct*s *out_edges* and *in_edges* are synonymous to arcs on an SDFG. The *in_edges* are incoming arcs to a block and *out_edges* are arcs that leave a block [Listing 5.4, Line 1 & 8]. Each arc has an arc label with the integer variable *arc_name*, their respective production and consumption rates and a pointer of type *edges* either to another SDF block or from an SDF block, depending on whether it is an incoming or outgoing arc.

Listing 5.4. struct out_edges & in_edges

```
1 struct out_edges{
2         int prod_rate;  // production rate
3         int arc_name;   // arc label
4         edges* to_ptr;  // pointer to next block
5         int delay;      // delay on this arc
6 };
7
8 struct in_edges {
9         int cons_rate;
10        int arc_name;
11        edges* from_ptr;
12        int delay;
13 };
```

The *struct* and *class* definitions in Listing 5.4 allow us to define an SDF block shown in Figure 5.3.

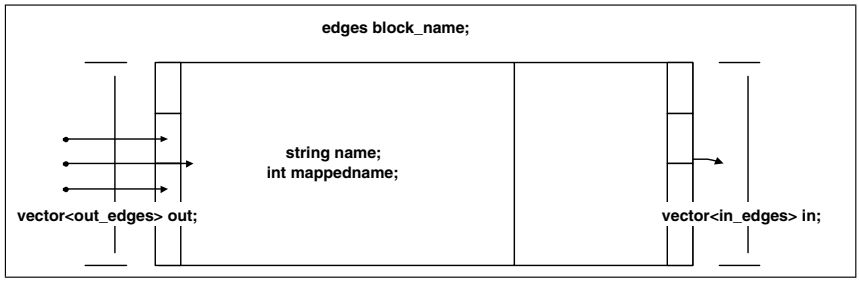


Figure 5.3. Synchronous Data Flow Block.

This representation of an SDF block is instantiated from within an *SC_MODULE()*. This makes an *SC_MODULE()* to be of type SDF method process. This means that one SDF block can only be represented by one *SC_MODULE()*.

We continue to explain the modeling style needed when modeling with the SDF kernel later in this chapter. We also describe the function calls that are required to create an SDF model. We present the prerequisites for the SDF kernel such as the linear Diophantine equation solver, creating a *repetition vector* from the solver and using it to construct a *firing order* yielding an *executable schedule* based on algorithms in [5, 11].

3. Scheduling of SDF

3.1 Repetition vector: Linear Diophantine Equations

The first issue of creating an *executable schedule* is discussed in [38] where Lee et al. describe a method whereby static scheduling of these

function blocks can be computed during compile time rather than run time. The method utilizes the predefined consumption and production rates to construct a set of linear Diophantine [11] homogeneous system of equations and represent it in the form of a topology matrix Γ. It was shown in [38] that in order to have a solution, Γ must be of rank $s-1$ where s is the number of blocks in the SDFG. Solution to this system of equations results in a *repetition vector* for the SDFG. An algorithm used to compute Hilbert's basis [11] solves linear Diophantine equations using the Completion procedure [11, 51] to provide an integer-valued Hilbert's basis. However, the fact that the rank is $s-1$ shows that for SDFs the Hilbert's basis is uni-dimensional and contains only one basis element [38].

Solving linear Diophantine equations is crucial in obtaining a valid *repetition vector* for any SDF graph. A tidy mechanism using the production and consumption rates to construct 2-variable equations and solving this system of equations results in the *repetition vector*. The equations have 2-variables because an arc can only be connected to two blocks. Though this may seem as a simple problem, the simplicity of the problem is challenged with the possibility of the solution of Diophantine equations coming from a real-valued set. This real-valued set of solutions for the Diophantine equations is unacceptable for the purpose of SDF since the number of firings of the blocks require being integral values. Not only do the values have to be integers, but they also have to be non-negative and non-zero, since a strongly connected SDFG can not have a block that is never fired. These systems of equations in mathematics are referred to as linear Diophantine equations and we discuss an algorithmic approach via the Completion procedure with an added heuristic to create the *repetition vector* as presented in [11].

We begin by defining a system of equations parameterized by $\overrightarrow{a} = \{a_i | i = 1...m\}$, $\overrightarrow{b} = \{b_j | j = 1...n\}$ and $\{c, m, n \in \mathbb{N}$ such that the general form for an inhomogeneous linear Diophantine equation is:

$$a_1 x_1 + ... + a_m x_m - b_1 y_1 - ... - b_n y_n = c \tag{5.1}$$

and for a homogeneous Diophantine equation is:

$$a_1 x_1 + ... + a_m x_m - b_1 y_1 - ... - b_n y_n = 0 \tag{5.2}$$

where only integer valued solutions for $\overrightarrow{x}$ and $\overrightarrow{y}$ are allowed. Continuing with the example from Figure 5.1, the arc going from block A to block B via arc label 1 results in Equation 5.3, where u, v, w, x, y and

z represent the number of times blocks A, B, C, D, E, F, and G have to be fired respectively. We refer to the producing block as the block providing the arc with a token and the consuming block as the block accepting the token from the same arc. For arc label 1, the consuming and producing blocks are block B and block A respectively. Therefore, for every arc, the equation is constructed by multiplying the required number of firings of the producing block with the production rate subtracted by the multiplication of the required number of firings of the consuming block with the consumption rate and setting this to be equal to 0.

$$1u - 2v + 0w + 0x + 0y + 0z = 0 \tag{5.3}$$

For the entire SDFG shown in Figure 5.1, the system of equations is described in Equations 5.4. Note that this is a homogeneous system of equations in which the total number of tokens inserted into the system equals the total number of tokens consumed. Our SDF scheduling implementation requires only homogeneous linear Diophantine equations, hence limiting our discussion to only homogeneous Diophantine equations.

$$\begin{aligned}
1u - 2v + 0w + 0x + 0y + 0z &= 0 \\
1u + 0v - 2w + 0x + 0y + 0z &= 0 \\
0u + 1v + 0w + 0x - 1y + 0z &= 0 \\
0u - 1v + 0w + 1x + 0y + 0z &= 0 \\
0u + 0v + 2w - 2x + 0y + 0z &= 0 \\
0u + 0v + 3w + 0x + 0y - 1z &= 0 \\
0u + 0v + 0w + 0x + 3y - 1z &= 0
\end{aligned} \tag{5.4}$$

This system of equations as you notice are only 2-variable equations yielding the topology matrix Γ as:

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

Solving for $\overrightarrow{X}$ in $\Gamma\overrightarrow{X} = 0$ yields the *repetition vector* for the SDFG in Figure 5.1. A linear Diophantine equation solver [51] solves these

topology matrices for system of equations such as in Equations 5.4. The results from the solver for Equations 5.4 are shown in Table 5.1.

Table 5.1. Results from Diophantine Solver

A=u	B=v	C=w	D=x	E=y	F=z
2	1	1	1	1	3

Notice that this methodology of creating a *repetition vector* is specific for acyclic SDF graphs. We discuss SDF graphs with cycles and producing *repetition vector*s for them later in this chapter.

3.2 Linear Diophantine Equation Solver

The problem of solving a system of equations for non-zero, minimal integer-valued solutions was addressed by many mathematicians such as Huet in 1978 [30], Fortenbacher in 1983 [18], Guckenbiehl & Herold in 1985 [26] etc. Of these, A. Fortenbacher and M. Clausen [11] introduce a lexicographic algorithm, which they called the Completion procedure algorithm. We limit our discussion to the Completion procedure algorithm since the labelled digraph approach they discussed is simply an extension of the same concept using labelled digraphs to visualize the problem.

Beginning with some notation, take the general form of a linear inhomogeneous Diophantine equation in Equation 5.1 where *S(a, b, c)* is the set of all nonnegative integer solutions and rewrite it such that the solution set $(\overrightarrow{\xi}, \overrightarrow{\eta}) \in \mathbb{N}^{m+n}$ satisfies the inhomogeneous Diophantine equation $\Sigma_i a_i \xi_i$ - $\Sigma_j b_j \eta_j = c$ for $\{i, j\} \in \mathbb{N}$. Evaluation of the left-hand side of the Diophantine equation is termed as the *defect* at a certain point from the set of $(\overrightarrow{\xi}, \overrightarrow{\eta}) \in \mathbb{N}^{m+n}$ [11]. So, the *defect* of $(\overrightarrow{\xi}, \overrightarrow{\eta})$ is evaluated by $d((\overrightarrow{\xi}, \overrightarrow{\eta})) := \Sigma_i a_i \xi_i$ - $\Sigma_j b_j \eta_j$ where a solution to the equation yields $d((\overrightarrow{\xi}, \overrightarrow{\eta})) = c$. For homogeneous Diophantine equations the equations are similar except that $c = 0$.

The Completion algorithm provided by Fortenbacher and Clausen [11] begins by creating three sets *P, M and Q* representing the set of proposals, the set of minimal solutions and a temporary set, respectively, as shown in Algorithm 5.1. A proposal is the first minimum guess by the algorithm for computing Hilbert's basis. The initialization of P starts with the minimal proposals that are used through the completion procedure. The other two sets M and Q are initially empty sets. The algorithm begins by selecting a proposal P_1 and during the Completion procedure it

increments this proposal according to the *defect* of that proposal. For a proposal $p = (\overrightarrow{\xi}, \overrightarrow{\eta})$, if $d(p) < 0$ then $\overrightarrow{\xi}$ is incremented, and if $d(p) > 0$ then $\overrightarrow{\eta}$ is incremented. If $d(p) = 0$ then a minimal solution is found and this is added to M. A test for minimality is performed and proposals that are not minimal with respect to the computed solution are removed from P. Once there are no more proposals the algorithm terminates. A Pascal implementation was provided in the paper that was converted to a C implementation by Dmitrii Pasechnik [51]. We further converted the C implementation to a C++ implementation

Algorithm 5.1: Completion procedure [11]

{Initialization}
$P_1 := (e_1, \overrightarrow{0}), ..., (e_m, \overrightarrow{0})$
$M_1 := NULL$
$Q_1 := NULL$
{Completion step}
$Q_{k+1} := \{p + (\overrightarrow{0}, e_j) | p \in P_k, d(p) > 0, 1 \leq j \leq n\}$
$\quad \cup \{p + (e_1, \overrightarrow{0}) | p \in P_k, d(p) < 0, 1 \leq i \leq m\}$
$M_{k+1} := \{p \in Q_{k+1} | d(p) = 0\}$
$P_{k+1} := \{p \in Q_{k+1} \setminus M_{k+1} |\ p\ \ minimal\ \ in\ \ p\ \cup\ \bigcup_{i=1}^{k} M_i\}$
{ Termination }
$P_k = NULL?$
$M := \bigcup_{i=1}^{k} M_i$

For our implementation, we employ the same algorithm to solve Diophantine equations with an added heuristic specific for SDFGs. We work through the running example presented in Figure 5.1 to show the added heuristic. Let us first explain why there is a need for a heuristic. Figure 5.1 contains seven equations and six unknowns, over-constraining the system of equations, but Algorithm 5.1 does not explicitly handle more than one equation. Hence, there has to be a way in which all equations can be considered at once on which the Completion procedure is performed. One may speculate an approach where all the equations are added and the *defect* of the sum of all equations is used to perform the algorithm. However, this is incorrect since the *defect* of the sum of the equations can be zero without guaranteeing that the *defect* for the individual equations being zero. This case is demonstrated in Step 3 in Table 5.2. This occurs because the set of solutions when considering the sum of all equations is larger than the set of solutions of the system of equations as described in seven equations, which can be confirmed by taking the rank of the matrices that the corresponding equations yield. Hence, we provide a heuristic that ensures a correct solution for the system of equations.

Algorithm 5.2: Completion procedure with Heuristic

Given m simultaneous 2 variable homogeneous Diophantine equations on a set of n variables, this algorithm finds a solution if one exists.
Let $E = \{(eq_1, eq_2, ..., eq_m)\}$ be the equation set.
Let P be an n tuple of positive integers initially $P := \{\overrightarrow{1}\}$
Let $x_1, x_2, ..., x_n$ be a set of variables.
Let $eq_j \equiv a_j^1 x_{l_j} - a_j^2 x_{k_j} = 0$ where $(j = 1, 2, ..., m)$ and a_j^1 , a_j^2 are coefficients of the j^{th} equation eq_j where $VARS(eq_j) = \{x_l, x_k\}$
Let $rhs(eq_j) = a_j^1 x_{l_j} - a_j^2 x_{k_j}$ which is a function of $\{x_l, x_k\}$
Let $INDICES(eq_j) = \{l_j, k_j\}$
Let d^j be the defect of eq_j evaluated at (α, β) such that $d^j = d(eq_j, \alpha, \beta) = (rhs(eq_j))|_{x_{l_j} \mapsto \alpha, x_{k_j} \mapsto \beta}$ where for any function over two variables u, v, $f(u,v)|_{u \mapsto a, v \mapsto b} = f(a,b)$
$D = \{d(eq_j, p_{l_j}, p_{k_j}) \mid VARS(eq_j) = \{l_j, k_j\}\}$ {So D is an m tuple of m integers called the defect vector}
$MAXINDEX(D) = j$ where $j = min\{r \mid d(eq_r, p_{l_r}, p_{k_r}) \geq d(eq_n, p_{l_n}, p_{k_n}) \, \forall \, n \in \{1, 2, ..., m\}$ *and* $\{l_n, k_n\} = INDICES(eq_n)\}$
{Repeat until defect vector D is all zeros or the maximum proposal in P is less then or equal to the lowest-common multiple of all the coefficients in all equations}
while $((D! = \overrightarrow{0}) \bigwedge (max(\overrightarrow{p}) \leq (lcm_{(r=1,2),(j=1..m)}(a_j^r)))$ **do**
  $j = MAXINDEX(D)$
  if $(d^j < 0)$ **then**
    Let x be the l^{th} variable and
    $(x, y) = VARS(eq_j)$
    for *all* $eq_i \in E$ **do**
      if $(x \in \{VARS(eq_i)\})$ **then**
        Re-evaluate d^i with $[p_l \mapsto p_l + 1]$
      end if
    end for
    Update $p_l \in P$ with $[p_l \mapsto p_l + 1]$
  else
    if $(d^j > 0)$ **then**
      Let y be the k^{th} variable and
      $(x, y) = VARS(eq_j)$
      for *all* $eq_i \in E$ **do**
        if $(y \in \{VARS(eq_i)\})$ **then**
          Re-evaluate d^i with $[p_k \mapsto p_k + 1]$
        end if
      end for
      Update $p_k \in P$ with $[p_k \mapsto p_k + 1]$
    end if
  end if
end while

The heuristic we implement ensures that before processing the *defect* of the proposal, the proposal is a vector of ones. By doing this, we indicate that every function block in the SDFG has to fire at least once.

After having done that, we begin the Completion procedure by calculating the *defect* of each individual equation in the system of equations and recording these values in a *defect* vector D. Taking the maximum *defect* from D defined as d^j for that j^{th} equation, if $d^j > 0$ then the correct variable for j is incremented and if $d^j < 0$, then the appropriate variable for the j is incremented for the j^{th} equation. However, updating the *defect* d^j for that equation is not sufficient because there might be other occurrences of that variable in other equations whose values also require being updated. Therefore, we update all occurrences of the variable in all equations and recompute the $defect$ for each equation. We perform this by checking if the updated variable exists in the set of variables for every equation extracted by the *VARS()* sub-procedure and if that is true, that equation is re-evaluated. The algorithm repeats until the *defect* vector $D = \overrightarrow{0}$ terminating the algorithm. The algorithm also terminates when the lowest-common multiple of all the coefficients is reached without making the $defect$ vector $D = \overrightarrow{0}$ [11]. This is because none of the proposal values can be larger than the lowest-common multiple of all the coefficients[11]. For Figure 5.1, the linear Diophantine equations are as follows: $u - 2v = 0, u - 2w = 0, v - y = 0, v + x = 0, 2w - 2x = 0, 3w - z = 0, 3y - z = 0, -u - z = 0$ and the *repetition vector* from these equations is shown in Table 5.1

To further clarify how the algorithm functions, we walk through the example in Figure 5.1 and compute the *repetition vector*. We define the proposal vector P as $\overrightarrow{p} = (u, v, w, x, y, z)$ where the elements of $\overrightarrow{p}$ represent the number of firings for that particular block. Similarly, we define a $defect$ vector $\overrightarrow{d} = (eq_1, eq_2, eq_3, eq_4, eq_5, eq_6, eq_7)$ where eq_n is the n^{th} equation in the system of equations. The system of equations are restated below:

$$
\begin{aligned}
eq_1 &= 1u - 2v + 0w + 0x + 0y + 0z = 0 \\
eq_2 &= 1u + 0v - 2w + 0x + 0y + 0z = 0 \\
eq_3 &= 0u + 1v + 0w + 0x - 1y + 0z = 0 \\
eq_4 &= 0u - 1v + 0w + 1x + 0y + 0z = 0 \\
eq_5 &= 0u + 0v + 2w - 2x + 0y + 0z = 0 \\
eq_6 &= 0u + 0v + 3w + 0x + 0y - 1z = 0 \\
eq_7 &= 0u + 0v + 0w + 0x + 3y - 1z = 0
\end{aligned}
$$

and steps in processing this system of equations with Algorithm 5.2 are shown in Table 5.2.

In Table 5.2, the proposal vector begins as a vector of all ones with its computed *defect*. The next equations to consider are the ones with

Table 5.2. Solution steps for example using Completion procedure

Step	$\overrightarrow{p}$	$\overrightarrow{d}$
1	(1, 1, 1, 1, 1, 1, 1)	(-1, -1, 0, 0, 0, 2, 2)
2	(1, 1, 1, 1, 1, 1, 2)	(-1, -1, 0, 0, 0, 1, 1)
3	(1, 1, 1, 1, 1, 1, 3)	(-1, -1, 0, 0, 0, 0, 0)
4	(2, 1, 1, 1, 1, 1, 3)	(0, 0, 0, 0, 0, 0, 0, 0)

the highest individual defect. We perform the Completion procedure on equations with the highest *defect*, that are eq_6 & eq_7, and increment z twice, to reduce the defect of eq_6 & eq_7 to zero. Then, we reduce the negative $defect$s for eq_1 & eq_2, that we compensate by incrementing v. This results in $defect$ vector being all zeros completing the completion procedure and giving a *repetition vector*. Though this example is strictly for acyclic SDFGs, we use the same algorithm in solving cyclic SDFGs. We present our discussion and algorithms for creating *executable schedule*s for cyclic SDF graphs in the next section.

3.3 Firing Order: repetition vectors for non-trivial cyclic SDF graphs

Most DSP systems have feedback loops and since the SDF MoC is used for DSP, we expect occurrences of loops in SDFGs. These feedback loops are represented as cycles in an SDFG and it is necessary for our SDF kernel to be able to efficiently handle these types of cycles. In this section we explain how cycles affect the *repetition vector* and discuss the *firing order* produced by the algorithm developed in [5].

We previously described the algorithm used in creating a *repetition vector* and we employ the same algorithm in calculating the *repetition vector* for cyclical SDFGs. However, the order in which these blocks have to be fired needs to be computed. The *firing order* is important because the SDF paradigm requires a specific order in which the function blocks are executed. SystemC's DE kernel schedules its processes in an unspecified manner, so it could schedule block F from Figure 5.4 as the first block for execution, which is incorrect for the SDF model. The *repetition vector* only describes the number of times F needs to be fired and not in which order F is fired. Until the proper *firing order* is found the system would deadlock due to F not having enough input on the incoming arcs from blocks C and E. We can see that blocks C and E have to be fired before F can be fired to correctly execute the SDFG.

For acyclic graphs as shown in Figure 5.4, one can use a topological sorting via Depth-First Search (DFS) algorithm to compute the *firing order*. However, in the presence of a cycle, a topological ordering does

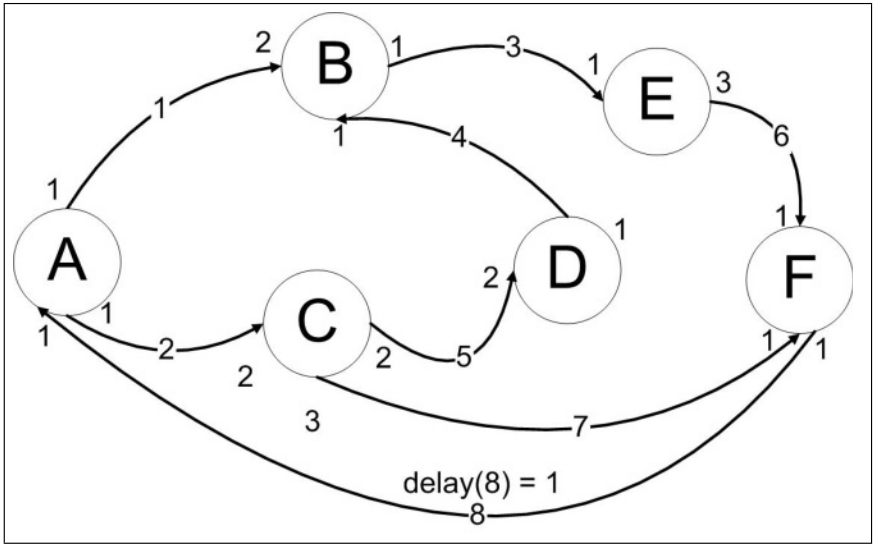


Figure 5.4. Example of a cyclic Synchronous Data Flow Graph [53].

not exist. Since, our goal is to obtain the *firing order* regardless of the SDFG containing cycles or not we employ another algorithm in constructing the *firing order*.

Bhattacharyya, Murthy and Lee in [5] developed scheduling algorithms for SDF of which one scheduling algorithm determines the *firing order* of non-trivial (cyclic or acyclic) SDFGs. However, before we present scheduling Algorithm 5.3, we discuss the *delay* terminology for an SDFG shown in Figure 5.4. Note the production rate at the head of the arcs, consumption rate at the tail of the arcs (marked by the arrow head), arc label or name in the middle of the arc and a newly introduced $delay(\alpha) = \gamma$ where α is an arc label and γ is the *delay* on that arc. *delay* [5] represents the number of tokens the designer has to inject initially into arc α for the SDF model to execute. The concept of *delay* is necessary when considering cyclical SDFGs. This is because of the additional constraint set by the SDF paradigm that every block only executes when it has sufficient input tokens on its input arcs, thus a cycle as indicated by arc label 8 in Figure 5.1 without the *delay* causes the SDFG to deadlock. Hence, no blocks can be fired. The *delay* acts as an initial token on that arc to allow simulation to begin. We introduce the concept of *delay* on all arcs, except we omit displaying the *delay*s that are zero, like on arc label 1 and 2 and so on.

Scheduling Algorithm 5.3 creates a *firing order* by using the *repetition vector* from the Hilbert's solver. If the *firing order* is valid then the SDF

kernel executes the SDF processes in the correct sequence for the correct number of times. The simulation terminates if either the *repetition vector* is invalid or the SDFG is inconsistent. An SDFG is inconsistent when the number of times every block scheduled in the *firing order* is not reflected by the number of times the block is supposed to be scheduled for execution as per the *repetition vector*.

Algorithm 5.3: Construct Valid Schedule Algorithm [5]
Require:
Let *ready* to be a queue of function block names
Let *queued* and *scheduled* to be vectors of non-negative integers indexed by the function blocks in SDFG
Let S be a schedule and initialize S to null schedule
Let **state** be a vector representing the number of tokens on each edge indexed by the edges
Let **rep** be a vector showing results from the Diophantine equation solver computed by Algorithm 5.2 (*repetition vector* indexed by the function block names)

for *all function blocks in SDFG* **do**
  for *each incoming edge* α **do**
    store delay on α to $state(\alpha)$
  end for

  for *each outgoing edge* β **do**
    store delay on β to $state(\beta)$
  end for
end for
for *each function block N in SDFG* **do**
  save **rep** for N in temporary variable *temp_rep*
  for *each incoming edge* α **do**
    set *del_cons* equal to $\lfloor delay(\alpha)/cons\_rate(\alpha) \rfloor$
    $temp\_rep = min(temp\_rep, del\_cons)$
  end for
  if $temp\_rep > 0$ **then**
    store *temp_rep* in $queued(N)$
    store 0 in $scheduled(N)$
  end if
end for

while *ready* is not empty **do**
  pop function block N from *ready*
  add $queued(N)$ invocations to S
  increment $scheduled(N)$ by value of $queued(N)$
  store *temp_n* to $queued(N)$
  set $queued(N)$ value to 0
  for *each incoming edge* α *of function block* N **do**
    set **state** for α is decremented by ($temp\_n \times cons\_rate$ on (α))
  end for

```
    for each outgoing edge β of function block N do
      set state for β is incremented by (n × prod_rate on (β))
    end for
    for each ougoing edge α of function block N do
      to_node is the function block pointed by α
      temp_r = subtract rep for to_node by scheduled(to_node)
    end for
    for each incoming edge γ of to_node(α) do
      set del_cons to ⌊state value for γ / cons_rate on γ⌋
    end for
    if (temp_r > queued(to_node)) then
      push to_node to ready
      set queued(to_node) to temp_r
    end if
  end while
  for each function block N in SDFG do
    if (scheduled vector for N ≠ to rep for N) then
      Error::Inconsistent Graph
      Exit
    end if
  end for
  S contains schedule
```

In Algorithm 5.3, ***state*** is a vector that initially contains the *delay*s on each of the arcs in the SDFG and is indexed by the arc names labelled α or β. During execution, ***state*** denotes the number of tokens on each arc. Similarly, *queued* and *scheduled* are vectors indexed by function block name N for the purpose of storing the number of times the block is *scheduled* and the number of times a block has to be scheduled to be fired is *queued*, respectively. *rep* is a temporary pointer to point to the *repetition vector* and *ready* holds the blocks that can be fired. The algorithm iteratively determines whether the SDFG is consistent and if so, results in an *executable schedule*.

The initialization begins by first traversing through all the function blocks in the SDFG and setting up the ***state*** vector with its corresponding *delay* for all the arcs on every block. Once this is done, every block is again traversed and for every incoming arc, the minimum between the *repetition vector* for that block and the $delay(\alpha)/cons\_rate(\alpha)$ is sought. If this minimum is larger than zero, then this block needs to be scheduled and added to *ready* since that means it has sufficient tokens on the incoming arcs to fire at least once. This initialization also distinguishes the blocks that are connected in cycles with sufficient *delay* values and schedules them. Scheduling of the remaining blocks is performed in a similar fashion with slight variation.

If the SDFG is consistent, the first block from *ready* is popped and appended to the *schedule* for the number of times the block is to be invoked. For all the incoming edges of this block the *delay* for this arc is recalculated. For the outgoing edges a similar calculation is done except this time the ***state*** for the arc is incremented by the production rate multiplied by *temp_n*. Basically, the algorithm looks at all the outgoing edges and the blocks pointed by these outgoing edges and proceeds to traverse focusing on all blocks pointed by the outgoing edges of the block just popped of the *ready*. Finally, a check for inconsistency is performed where the number of times each block is scheduled has to be equivalent to the number of times it is supposed to be fired from the *repetition vector*. The algorithm concludes with the correct schedule in S for a consistent SDFG.

This scheduling algorithm yields the schedule in the correct *firing order* with the number of times it is to be fired. The kernel will fire according to this schedule. Acyclic and cyclic SDFGs are handled correctly by this algorithm. The final *executable schedule* is stored in *sdf_schedule* accessible to the kernel for execution.

4. SDF Modeling Guidelines

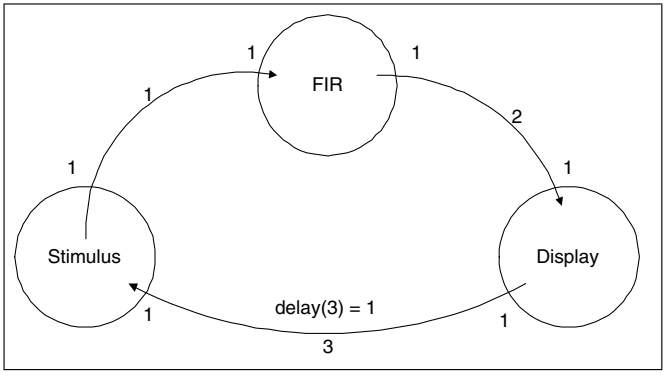


Figure 5.5. FIR Example of a Synchronous Data Flow Graph [53].

Efforts in reducing difficulty of modeling other MoCs have been a key consideration in implementing the SDF kernel. Though, we believe that we have reduced the level of difficulty in designing SDF models in SystemC by increasing the modeling fidelity, we also provide guidelines in creating SDF models. A simple example with full source code segments demonstrates the style. We use the FIR example from Figure 5.5 to compare the FIR example provided with the SystemC 2.0.1 distribution modeled with the DE kernel (that has already been shown in Chapter

3) with the converted model using the SDF kernel. We edit the source code to remove some *std :: cout* statements to make the output from DE FIR example to match the output of the SDF FIR model, but the functionality of the FIR in DE and SDF remains the same. *fir_const.h* is not included in the listings since it is a list of constants that will be available in the full source prints at [36].

Listing 5.5. stimulus.h

```
1 #include <queue>
2 extern sdf_graph sdf1;
3 using namespace std;
4
5 SC_MODULE(stimulus) {
6   edges stimulus_edge;
7   sc_int<8>      send_value1;
8   SC_CTOR(stimulus)  {
9     stimulus_edge.set_name(name(), sdf1.sdflist);
10    SC_METHOD(entry);
11      send_value1 = 0;
12   }
13  void entry();
14 };
```

Notice in Listing 5.5 that the Stimulus block has no input or output ports along or control signal ports. This refers to declarations of ports using *sc_in<...>* or *sc_out<...>*. These are no longer required in an SDF model because static scheduling does not require control signals and our method of data passing is through STL *queue<...>* structures. We do not strictly enforce their removal and if the designer pleases to use SystemC channels and ports for data paths then that can also be employed. However, the SDF kernel statically schedules the SDF blocks for execution at compile time, hence there is no need for one block to signal to the following block when data is ready to be passed on, obviating the need for control signals. For data paths, using signals and channels in SystemC generate events reducing simulation efficiency. Our advised approach is to use *queue<...>* STL queues to transfer data within the SDF model instead of using SystemC. Only the data that has to be passed from one block to another requires an instantiation of a queue such as the *extern queue<int> stimulusQ* [Listing 5.6, Line 6]. Instantiation of *stimulus_edge* object is crucial in defining this *SC_MODULE()* as an SDF method process [Listing 5.5, Line 5 & 6]. This object is used to pass the name of the *SC_MODULE()* and the SDFG which this block belongs to as shown in [Listing 5.5, Line 9].

The queues used in the FIR model are *stimulusQ* and *firQ*, where *stimulusQ* connects the Stimulus block to the FIR block and *firQ* connects from the FIR block to the Display [Listing 5.8, Line 4 & 5]. However,

Listing 5.6. stimulus.cpp

```
#include <systemc.h>
#include "stimulus.h"
#include <queue>

using namespace std;
extern queue<int> stimulusQ;

void stimulus::entry() {
  stimulusQ.push(send_value1);
  send_value1++;
}
```

instantiation of these queues has to be done in a particular manner. Since every arc connects two blocks implies that two blocks must have the same queue visible to them. So, *stimulusQ* should be accessible by the Stimulus block as well as the FIR block. The easiest approach is to instantiate these queues globally in the toplevel file and make them *extern* keyword when a block needs to refer to the *queue<...>*s (if in a different file). With this method all the files must be a part of one compilation since *extern* informs the compiler that the particular variable (in this case *queue<...>*) is instantiated in some other file external to the current scope. Furthermore, memory is not allocated when *extern* keyword is encountered because the compiler assumes that the variables with *extern* keyword have been properly defined elsewhere.

Listing 5.7. fir.h

```
#include <queue>
extern sdf_graph sdf1;
using namespace std;

SC_MODULE(fir) {
  sc_int<9> coefs[16];
  sc_int<8> sample_tmp;
  sc_int<17> pro;
  sc_int<19> acc;
  sc_int<8> shift[16];
  edges fir_edge;
  SC_CTOR(fir)  {
      fir_edge.set_name(name() /*"process_body"*/, sdf1.sdflist)
          ;
      SC_METHOD(entry);
      #include "fir_const.h"
      for (int i = 0; i < 15; i++)  {
            shift[i] = 0;
      }
  void entry()
  };
```

Every SDF block (SDF *SC_MODULE()*) must also have access to the SDFG that it is to be inserted in. This means the *sdf_graph* object that is instantiated globally must be available to the *SC_MODULE()*s such as in [Listing 5.7, Line 2]. However, the integral part of the *SC_MODULE()* declaration is the instantiation of the *edges* object as shown in [Listing 5.7, Line 11]. This object is accessed by the SDF kernel in determining certain characteristics during scheduling. These characteristics are set by member functions available in the *edges* class. One of the first member functions encountered in the *SC_MODULE()* declaration is the *set_name(...)* [Listing 5.7, Line 13] function that is responsible for providing the *edges* object with the module name and the storage list of SDF method processes. The *name()* function from within the *SC_MODULE()* returns the name of the current module. In [Listing 5.7, Line 13] *sdf1.sdflist* is the list where the addresses of the *SC_METHOD()* processes are stored for access by the SDF kernel. Apart from those alterations, the structure of an *SC_MODULE()* is similar to regular SystemC processes.

Listing 5.8. fir.cpp

```
1#include <systemc.h>
2#include "fir.h"
3
4extern queue<int> stimulusQ;
5extern queue<int> firQ;
6
7void fir::entry() {
8   sample_tmp = stimulusQ.front();   stimulusQ.pop();
9   acc = sample_tmp*coefs[0];
10  for(int i=14; i>=0; i--) {
11     pro = shift[i]*coefs[i+1];
12     acc += pro;
13  }
14  for(int i=14; i>=0; i--) {
15     shift[i+1] = shift[i];
16  }
17  shift[0] = sample_tmp;
18  firQ.push((int) acc);
19}
```

Note that the functions used to insert data onto the *queue*<...>s are STL functions *push()* and *pop()* [Listing 5.8, Line 8 & 18]. There is also no check for the number of tokens ready to be received by each block. Naturally, this is not required since we are statically scheduling the SDF blocks for an appropriate number of times according to their consumption and production rates. However, this burdens the designer with the responsibility of carefully inserting sufficient tokens on the *queue*<...>s to ensure the simulation does not attempt at using invalid data.

Listing 5.9. display.h

```
1#include <queue>
2extern sdf_graph sdf1;
3using namespace std;
4
5SC_MODULE(display) {
6   int i, tmp1;
7   edges display_edge;
8   SC_CTOR(display)
9     { display_edge.set_name(name() /*"display"*/, sdf1.sdflist);
10      SC_METHOD(entry);
11      i = 0;
12    }
13  void entry();
14};
```

Listing 5.10. display.cpp

```
1#include <systemc.h>
2#include "display.h"
3extern queue<int> firQ;
4void display::entry(){
5   tmp1 = firQ.front();   firQ.pop();
6   cout << tmp1 << endl;
7   i++;
8   if(i == 5000000) {
9     sc_stop();
10  };
11}
12// EOF
```

The SDF kernel requires the modeler to specify the terminating value as in the DE kernel example. This is similar to the termination situations posed in [23]. However, we define a period of SDF as a complete execution of the SDF. In this example since there is a cycle, every period is an execution of the SDF model. We halt the execution after a specified number of samples using the *sc_stop()* [Listing 5.10, Line 9] which tells the kernel that the modeler has requested termination of the simulation.

The toplevel *SC_METHOD()* process labelled *toplevel* in [Listing 5.12, Line 2] constructs the SDF graph encapsulating that entire SDF model inside it. The choice of the toplevel process can be of any SystemC type. The entry function has to be manipulated according to the process type since they continue to follow SystemC semantics. The construction of this module is straightforward whereby pointers to each of the SDF methods are member variables and are initialized to their corresponding objects in the constructor. The important step is in constructing the SDFG within the constructor (*SC_CTOR()*) since the constructor is only invoked once per instantiation of the object. The functions *set_prod(...)* and *set_cons(...)* set the arcs on the SDFG. Every *SC_MODULE()* de-

Listing 5.11. main.cpp

```
1#include <systemc.h>
2#include "stimulus.h"
3#include "display.h"
4#include "fir.h"
5sdf_graph sdf1;
6queue <int> stimulusQ;
7queue <int> firQ;
8// General METHOD process to verify proper execution of ordinary
      DE METHODs
9SC_MODULE(foo) {
10   sc_in_clk clock;
11   void message()  {
12     cout << sc_time_stamp() << " foo with next_trigger executed
           " << endl;
13     next_trigger(2, SC_NS);
14   }
15     SC_CTOR(foo){
16       SC_METHOD(message)  {
17         sensitive << clock.pos();
18         };
19   }
20};
21// General CTHREAD process to verify proper execution of
      ordinary DE THREADs
22SC_MODULE(foo_cthread)  {
23   sc_out<bool> data_sdf;
24   sc_in_clk clock;
25   void msg () {
26     bool b= false;
27     while(1)  {
28       cout << sc_time_stamp() << " CTHREAD executed " << endl;
29       wait(3);
30       cout << "Instruct SDF to fire" << endl;
31       if (b == true)
32         b = false;
33       else
34         b = true;
35       data_sdf.write(b);
36     }
37   }
38   SC_CTOR(foo_cthread) {
39     SC_CTHREAD(msg, clock.pos()) {
40       sensitive << clock.pos() ;
41       };
42     }
43};
```

fines an SDF block that requires the arcs being set. The arguments of these set functions are: the address of the *edges* object instantiated in a module that it is pointed to or from, the production or consumption rate depending on which function is called and the *delay*. We also enforce a global instantiation of *sdf_graph* [Listing 5.11, Line 5] type object for every SDFG that is present in the model. Using the *schedule* class and the *add_sdf_graph(...)*, the SDFG is added into a list that is visible by the overlaying SDF kernel [Listing 5.12, Line 23]. In addition, this toplevel process must have an entry function that calls *sdf_trigger()* [Listing 5.12,

Listing 5.12. toplevel and main()

```
1// Top Level METHOD encapsulating the SDFG
2SC_MODULE(toplevel) {
3 sc_in<bool> data;
4  fir* fir1; //( "process_body*");
5  display* display1 ; //( "display");
6  stimulus* stimulus1; //("stimulus_block");
7  void entry_sdf()  {
8    sdf_trigger(name());
9  }
10  SC_CTOR(toplevel) {
11    SC_METHOD(entry_sdf)
12      sensitive << data ;
13
14    fir1 = new fir("process_body*");
15    display1 = new display("display");
16    stimulus1 = new stimulus("stimulus");
17    stimulus1->stimulus_edge.set_prod(&fir1->fir_edge,1 , 0);
18    fir1->fir_edge.set_cons(&stimulus1->stimulus_edge, 1, 0);
19    fir1->fir_edge.set_prod(&display1->display_edge, 1, 0);
20    display1->display_edge.set_cons(&fir1->fir_edge, 1, 0);
21    display1->display_edge.set_prod(&stimulus1->stimulus_edge
          , 1, 1);
22    stimulus1->stimulus_edge.set_cons(&display1->display_edge
          , 1, 1);
23    schedule::add_sdf_graph(name(), &sdf1);
24  }
25};
26
27int sc_main (int argc , char *argv[]) {
28  sc_clock          clock;
29  sc_signal<bool> data;
30  toplevel top_level("top_level*sdf*");
31  top_level.data(data);
32  foo foobar("foobar");
33  foobar.clock(clock);
34  foo_cthread foo_c("foo_C");
35  foo_c.clock(clock);
36  foo_c.data_sdf(data);
37  sc_start(-1);
38 return 0;
39}
```

Line 8] signalling the kernel to process all the SDF methods corresponding to this toplevel SDF module. These guidelines enable the modeler to allow for heterogeneity in the models since the toplevel process can be sensitive to any signal that is to fire the SDF. The *SC_CTHREAD()* [Listing 5.11, Line 39] partakes in this particular role where every three cycles the SDF model is triggered through the signal *data*. However, the designer has to be careful during multi-MoC modeling due to the transfer of data from the DE blocks to the SDF blocks. This is because there is no functionality in the SDF kernel or for that matter even the DE kernel that verifies that data on an STL *queue*<...> path is available before triggering the SDF method process. This has to be carefully handled by the designer. These style guides for SDF are natural to the paradigm

and we believe that this brief explanation of modeling in SDF provides sufficient exposure in using this SDF kernel along with the existing DE kernel.

4.1 Summary of Designer Guidelines for SDF implementation

The designer must remember the following when constructing an SDF model:

- To represent each *SC_MODULE()* as a single SDF function block as in Listing 5.5.
- Ensure that each process type is of *SC_METHOD()* process [Listing 5.5, Line 10].
- The module must have access to the instance of *sdf_graph* that it is to be inserted in [Listing 5.7, Line 2].
- An object of *edges* is instantiated as a member of the *SC_MODULE()* and the *set_name()* function is called with the correct arguments [Listing 5.7, Line 11 & 13].
- The instance of *sdf_graph* is added into the SDFG kernel list by calling the *static add_sdf_graph()* function from *schedule* class as in [Listing 5.12, Line 23].
- Set *delay* values appropriately for the input and output samples on the *queue*<...> channels for every arc [Listing 5.12, Line 22].
- Introduce a *sc_clock* object to support the timed DE MoC [Listing 5.12, Line 28].
- Ensure that the entire SDFG is encapsulated in a toplevel process of any type [Listing 5.12, Line 2 - 39].
- Toplevel process must have an entry function that calls *sdf_trigger()* ensuring that when this process is fired, its corresponding SDF processes are also executed [Listing 5.12, Line 8].

5. SDF Kernel in SystemC

We implement the SDF kernel and update the DE kernel function calls through the use of our API discussed in Chapter 8. We limit our alterations to the original source, but some change in original source code is unavoidable. Our approach for kernel implementation is in a particular manner where the SDF kernel exists within the Discrete-Event kernel.

Listing 5.13. split_processes() function from API class

```
void sc_domains::split_processes() {
  sc_process_table* m_process_table = de_kernel->
      get_process_table();
  const sc_method_vec& method_vec = m_process_table->method_vec
      ();
  if (sdf_domain.size() != 0) {
    for (int sdf_graphs = 0; sdf_graphs < (signed) sdf_domain.
        size() ; sdf_graphs++) {
      // Extract the address of SDFG
      sdf_graph * process_sdf_graph = sdf_domain[sdf_graphs];
      sdf* process_sdf = &process_sdf_graph->sdflist;
      for( int i = 0; i < method_vec.size() ; i++ ) {
        sc_method_handle p_method_h;
        sc_method_handle method_h = method_vec[i];
        if( method_h->do_initialize() ) {
          string m_name = method_h->name();
          bool found_name = false;
          for (int j = 0; j< (signed) process_sdf->size(); j++)
              {
            edges* edge_ptr = (*process_sdf)[j];
            if (strstr(method_h->name(), edge_ptr->name.c_str())
                != NULL ) {
              found_name = true;
              p_method_h = method_h;
            }/* END IF */
          }/* END FOR */
          // Check to see if the name of the current process is
              in the SDF list and route
            // accordingly to the correct METHODs list.
          if (found_name == true) {
            process_sdf_graph->sdf_method_handles.push_back(
                p_method_h);
            found_name = false;
          }
          else {
            // This must be a DE process - will be inserted
                itself
          }/* END IF-ELSE */
        }/* END IF */
      }/* END FOR */
      // remove the method handles added to SDF list from DE
          list
      for (int k = 0; k < (signed) process_sdf_graph->
          sdf_method_handles.size() ; k++) {
        sc_method_handle del_method_h = process_sdf_graph->
            sdf_method_handles[k];
        m_process_table->remove(del_method_h)->name();
      }/* END FOR */
    }
  } else {
    schedule::err_msg("NO SDF GRAPHS TO SPLIT PROCESS", "WW");
  }/* END IF-ELSE */
}/* END split_processes */
```

This implies that execution of SDF processes is performed from the DE kernel making the DE kernel the parent kernel supporting the offspring SDF kernel. We employ the parent-offspring terminology to suggest that SystemC is an event-driven modeling and simulation framework through which we establish the SDF kernel. However, this does not

mean that a DE model can not be present within an SDF model, though careful programming is required in ensuring the DE block is contained within one SDF block. For the future extension we are working on a more generic design for hierarchical kernels through an evolved API. Algorithm 5.4 displays the altered DE. The noticeable change in the kernel is the separation of initialization roles. We find it necessary to separate what we consider two distinct initialization roles as:

- Preparing model for simulation in terms of instantiating objects, setting flags, etc.
- Pushing processes (all types) onto the *runnable* queues and executing *crunch()* (see Chapter 3) once.

If there are manipulations required to the runnable process lists prior to inserting all the processes onto the *runnable* queues for execution during initialization, then the separation in initialization is necessary. For example, for the SDF kernel we require the processes that are SDF methods to be separated and not available to be pushed onto the runnable queues. This separation is performed using the *split_processes()* in the API class as shown in Figure 5.13, that identifies SDF methods and removes them from the list that holds all *SC_METHOD()* processes. This is not possible if the original initialization function for the DE kernel is unchanged because it makes all processes runnable during initialization. We are fully aware of the implications of this implementation in that it departs from the SystemC standard. However, the SystemC standard does not dictate how a Synchronous Data Flow kernel or programming paradigm is to behave, hence we feel comfortable in implementing such changes as long as the DE kernel concurs with SystemC standards. Another difference of the standard is allowing the designer to specify the order of execution of processes through an overloaded *sc_start(...)* function call. If the user has prior knowledge of a certain order in which the system, especially in the case of DE and SDF heterogeneous models, then the user should have flexibility in allowing for definition of order instead of using control signals to force order. The limitations of the SystemC standard will progressively become apparent once more MoCs are implemented resulting in more dissonance between SystemC standards and the goal of a heterogeneous modeling and simulation framework.

In addition to the separation of *SC_METHOD()* processes, the *crunch()* function that executes all the processes is slightly altered in execution of the processes. If a modeler specifies the order in which to fire the processes, then this order needs to be followed by the kernel. This requires popping all the processes from its respective runnable list

and storing them separately onto a temporary queue. These processes are then selected from this temporary queue and executed according to an order if it is specified, after which the remaining processes are executed. The need for this is to enable support for signal updates and *next_trigger()* to function correctly. When a process is sensitive to a signal and an event occurs on the signal, then this process can be *ready-to-run* causing it to be pushed onto the runnable list. If there is no ordering specified by the designer then the processes are popped regularly without requiring a temporary queue.

Likewise, the *simulate()* function suffered some alterations to incorporate one period of execution in a cycle. The *simulate()* function uses a *clock_count* variable to monitor the edges of the clock. This is necessary to enforce the SDF graph is executed once every cycle. To understand the need for it to execute once every cycle, we have to understand how an *sc_clock* object is instantiated. A clock has a positive and a negative edge and for the kernel the *sc_clock* creates two *SC_METHOD()* processes one with a positive edge and the second with a negative edge. These are then processed like normal *SC_METHOD()* processes, causing the *crunch()* function to be invoked twice. Therefore, the *clock_count* variable ensures that the SDF execution is only invoked once per cycle as per definition of a period. However, there is an interesting problem that this might result in when modeling in SDF. If there is to be a standalone SDF model, then there has to be an instance of *sc_clock* even if it is not connected to any ports in the SDF model. This is essential for the SDF graph to function correctly. We envision the SDF to be executed alongside with DE modules, hence it does not seem unnatural to expect this condition.

5.1 Specifying Execution Order for Processes

Providing the kernel with an execution order has to be carefully used by the modeler. This is because the semantics that belong to each process category (*SC_METHOD(), SC_THREAD(), SC_CTHREAD()*) are still followed. Hence, the blocks that trigger the SDF also adhere to the semantics, which can complicate execution when specifying order. Suppose an *SC_CTHREAD()*'s responsibility is to fire an SDF block and these are the only two blocks in the system such that they are called CTHREAD1 and SDF1. If the modeler specified the execution order as *"SDF1 CTHREAD1"* then the correct behavior will involve an execution of SDF1 followed by CTHREAD1 and then again SDF1. As expected, with flexibility comes complexity, but we believe allowing this kind of flexibility is necessary for modeler who understand how their model works.

5.2 SDF Kernel

The pseudo-code for the algorithm employed in altering the DE kernel to accept the SDF kernel is shown below.

Algorithm 5.4: DE Kernel and SDF Kernel
{classes edges, sdf_graph, schedule and sc_domains are already defined}

void de_initialize1()
perform update on primitive channel registries;
prepare all THREADs and CTHREADs for simulation;
{ **END initialize1()**}

void de_initialize2()
push METHOD handle onto regular DE METHOD runnable list;
push all THREADs onto THREAD runnable list;
process delta notifications;
execute crunch() to perform Evaluate-Update once.
{ **END initialize2()**}

void simulate()
initialize1();
if (clock count mod 2 == 0) **then**
  set run_sdf to true;
end if
execute crunch() until no timed notifications or runnable processes;
increment clock count;
{ **END simulate()**}

void crunch()
while (true) **do**
  if (there is a user specified order) **then**
    pop all methods and threads off runnable list and store into temporary
    while (parsed user specified order is valid) **do**
      find process handle in temporary lists and execute;
    end while
  else
    execute all remaining processes in the temporary lists;
  end if
  { Evaluate Phase }
  execute all THREAD/CTHREAD processes;
  break when no processes left to execute;
end while
{ Update Phase}
update primitive channel registries;
increment delta counter;
process delta notifications;
{ **END crunch()**}

SDF initialization is responsible for constructing an *executable schedule* for all SDFGs in the system. If any of the SDFGs is inconsistent then the simulation stops immediately, flagging that the simulation cannot be performed. This involves creating the input matrices for the Diophantine solver and creating an *executable* SDF schedule. A user calling the *sc_start(...)* function invokes the global function that in turn uses an instance of *sc_domains* that begins the initialization process of both the DE and SDF kernels as shown in Listing 5.14. *init_domain(...)* initializes the domains that exist in SystemC (SDF and DE so far) and then begins the simulation of the DE kernel (since SDF is written such that it is within the DE kernel). The *sdf_trigger(...)* global function is to be only called from the entry function of the toplevel SDF encapsulating the entire SDF model. This ensures execution of all SDF method processes specific for that SDFG.

Listing 5.14. Global functions

```
1 void sc_start( const sc_time& duration, string in )
2 {
3     model.init_domains(duration, in );
4     model.de_kernel->simulate(duration);
5 }/* END sc_start */
6
7 inline void sdf_trigger(string topname) {
8
9   if (model.sdf_domain.size() > 0)  {
10    // SDFG exists
11    model.sdf_trigger(topname);
12  }/* END IF */
13  else  {
14    schedule::err_msg("No SDFGs in current model, ensure
          add_sdf_graph() called ", "EE");
15    return;
16  }/* END IF-ELSE */
17 }/* END sdf_trigger */
```

The API class *sc_domains* described in Chapter 8 has function declarations to initialize the DE and SDF kernels implemented at the API level to invoke their respective DE or SDF functions. The *init_domains(...)* function shown in Listing 5.15 is responsible for initializing all the existing domains in SystemC. This function sets the user order string if specified then prepares the simulation in terms of instantiating objects and setting the simulation flags, splits the processes as explained earlier and readies the runnable queues. Followed by initialization of the SDF, which traverses through all SDFGs present in the system and creates an *executable schedule* for each one if one can be computed. Given that all conditions are satisfied and simulation is not halted during the scheduling process, the simulation begins.

Listing 5.15. init_domain() in sc_domains

```
1// initial the domains
2void sc_domains::init_domains(const sc_time & duration, string
      in ) {
3
4  if (in.size() > 0)
5    user_input(in);
6
7  // initialize the simulation flags
8  init_de();
9  // split the processes for every SDFG
10 split_processes();
11 // initialize the runnable lists
12 model.de_kernel->de_initialize2();
13 // initialize SDF for execution
14 init_sdf();
15}
```

Listing 5.16. sdf_trigger() and *find_sdf_graph()* in sc_domains

```
1void sc_domains::sdf_trigger(string topname)  {
2
3  string sdfname = topname+".";
4  sdf_graph * run_this;
5
6  for (int sdf_graphs = 0; sdf_graphs < (signed) model.
       sdf_domain.size(); sdf_graphs++) {
7    // pointer to a particular SDF graph
8    sdf_graph * process_sdf_graph = model.sdf_domain[sdf_graphs
         ];
9    if (strcmp(process_sdf_graph->prefix.c_str(), sdfname.c_str
         ())==0 ) {
10     run_this = process_sdf_graph;
11     if ( run_sdf == true){
12       // execute the SDF METHODs
13       run_this->sdf_simulate(sdfname);
14       run_sdf = false;
15     }/* END IF */
16   }/* END IF */
17 }/* END FOR */
18}/* END sdf_trigger */
19
20sdf_graph * sc_domains::find_sdf_graph(string sdf_prefix) {
21
22 for (int sdf_graphs = 0; sdf_graphs < (signed) model.
       sdf_domain.size(); sdf_graphs++) {
23   // pointer to a particular SDF graph
24   sdf_graph * process_sdf_graph = model.sdf_domain[sdf_graphs
         ];
25   if (strcmp(process_sdf_graph->prefix.c_str(), sdf_prefix.
         c_str())==0 )  {
26     return ( process_sdf_graph);
27   }/* END IF */
28 }/* END FOR */
29   return NULL;
30}/* END find_sdf_graph */
```

Listing 5.16 shows the definition of *sdf_trigger()* that calls the *sdf_simulate()* function responsible for finding the appropriate SDFG

(with the helper function *find_sdf_graph(...)* and executing the SDF processes corresponding to that SDFG.

The creation of the schedules is encapsulated in the *sdf_create_schedule(...)* function that constructs the topology matrix for the Diophantine equations solver, returns the solution from the solver and creates an *executable schedule* if one exists as demonstrated in Listing 5.17.

Listing 5.17. SDF initialization function

```
1 void sdf_graph::sdf_create_schedule() {
2
3   // Repeat for all the SDF graphs that are modelled
4   // Extract the address of first SDF
5   sdf_graph * process_sdf_graph = this;
6   int* input_matrix = schedule::create_schedule(
        process_sdf_graph);
7
8   if (input_matrix != NULL) {
9     hbs(process_sdf_graph->num_arcs, process_sdf_graph->
          num_nodes, input_matrix,&process_sdf_graph->result);
10
11    if ( process_sdf_graph->result != NULL) {
12      // Stored the address of the schedule in the resultlist.
13    }
14    else  {
15      schedule::err_msg("The result schedule is invalid. Halt
            simulation", "EE");
16      // Attempt at clean up for Matrix Input
17      free(input_matrix);
18      // Exit simulation
19      exit(1);
20    }/* END IF-ELSE */
21    free(input_matrix);
22  }/* END IF */
23  else {
24    schedule::err_msg("Input Matrix for SDF is invalid. Halt
          simulation", "EE");
25    exit(1);
26  }/* END IF-ELSE */
27  schedule::construct_valid_schedule(*process_sdf_graph);
28}/* END sdf_create_schedule */
```

The compilation of SystemC requires passing a compiler flag *_SDF_KERNEL_ENABLE*; hence the *#ifndef*s with that variable. The first stage is in creating a matrix representation that can be the input for the linear Diophantine equation solver. This is performed by traversing through all SDFGs that might exist in the system and passing a pointer to the SDFG as an argument to the *create_schedule(...)* function that creates the topology matrix. If the pointer to the topology matrix called *input_matrix* is valid (not *NULL*) then it is passed to the Diophantine equation solver via the *hbs*(...) call, which depending on whether there exists a solution returns a pointer with the result or returns *NULL*. The address of the matrix with the result is stored in that SDFG's *re-*

sult data member. Once the result is calculated it is checked whether it is not *NULL* and if it is then the simulation exits, otherwise the simulation proceeds by executing the *construct_valid_schedule(...)* function. Remember that the schedule class is a purely static class with all its member functions being static hence all function calls precede with *schedule::*. If at any point during the schedule construction either of the input to the Diophantine solver, or the result or even the scheduling for *firing order* finds inconsistencies then the simulation exits. The method of exit might not be the safest exit using the *exit(1)* function and an improvement might involve creating a better cleanup and exit mechanism or perhaps the use of *sc_stop()*.

Upon construction of the SDF *execution schedule*s the simulation begins. When a toplevel SDF process is identified, the user is required to invoke *sdf_trigger()* from its entry function leading to a brief discussion of that function shown in Listing 5.18. The underlying function called from within *sdf_trigger* is *sdf_crunch()* that actually executes the SDF processes as shown in Listing 5.18. This function traverses through the SDF method processes stored in this particular SDFG and executes them in the order created by the scheduling algorithm. It must be noted that these SDF processes are not pushed onto the runnable queues restricting their functionality by not allowing a safe usage of *next_trigger()* or the sensitivity lists.

The *sdf_simulate(...)* function simply calls the *sdf_crunch(...)* function as in Listing 5.19. The full source with changes in the original kernel to incorporate the SDF kernel is available at [36]. We have only discussed the main functions that are used to invoke the SDF kernel and point out to the reader to the few changes made to the original DE kernel to accommodate the SDF kernel. To summarize, we list the changes made to the existing kernel:

- Separation of initialization roles to allow for separation of processes.
- Splitting of *SC_METHOD()* processes. Not all *SC_METHOD()* processes are made runnable since we remove SDF block methods from the method process list.
- Specifying user order handles processes of all types. If a user order is specified then a temporary queue is used to pop the runnable lists, find the appropriate processes and execute them.
- Addition of a clock counter to invoke the SDF execution only once per cycle when the toplevel entry function is fired.

We believe these to be moderate changes given that we implement a completely new kernel working alongside with the DE kernel and accept

Listing 5.18. SDF crunch function

```
1 void sdf_graph::sdf_crunch(string sdf_prefix) {
2
3   sc_method_handle method_h;
4   #ifdef _SDF_KERNEL_ENABLE
5   /* Executes only the SDF methods that have been initialized.
        Ordinary DE based
6   METHODs will be executed later.
7   */
8   sdf_graph * process_this_sdf_graph = this; //find_sdf_graph(
        sdf_prefix);
9
10  if ( process_this_sdf_graph != NULL) {
11  // pointer to particular edges* list
12    sdf * process_sdf = &process_this_sdf_graph->sdf_schedule;
13
14    // traverse through all the scheduled nodes in that
          particular order
15    for (int j = 0; j < (signed) process_sdf->size(); j++)  {
16      edges* edg_ptr = (*process_sdf)[j];
17      for (int i = 0; i < (signed) process_this_sdf_graph->
            sdf_method_handles.size(); i++)  { //
            asdfasdfasfdasdfasd
18        method_h = process_this_sdf_graph->sdf_method_handles[i
            ];
19        if (strstr(method_h->name(), edg_ptr->name.c_str()))  {
20          try    {
21                method_h->execute();
22          }
23          catch( const sc_exception& ex )    {
24               cout << "\n" << ex.what() << endl;
25               return;
26          }
27        }/* END IF */
28      }/* END FOR */
29    }/* END FOR */
30  }
31  else {
32    schedule::err_msg("SDF Graph not found ", "EE");
33  }/* END IF-ELSE */
34  #endif
35}/* END sdf_crunch */
```

Listing 5.19. SDF simulate function

```
1 void sc_simcontext::simulate( const sc_time& duration ) {
2   // Execute for a period (one execution of SDF)
3   sdf_crunch(prefix);
4 }
```

that there can be better conceivable methods of implementing this as the C++ language allows an infinite number of implementations.

6. SDF Specific Examples

Experimentation of the SDF kernel has been an incremental process beginning with pure SDF models followed by heterogeneous models. To

evaluate the efficiency enhancement by our SDF kernel we set out to experiment with a few models that are amenable to SDF style modeling. In this section, we show the results of simulating the same models for three distinct modeling styles and different sample sizes of data. The first set of experiments are pure SDF models for the Finite Impulse Response (FIR), Fast Fourier Transform (FFT), and the Sobel edge detection algorithm [47]. The second set involves creating a combination of Discrete-Event and Synchronous Data Flow models shown in Chapter 9. In [62] around 50% or more improvement in simulation efficiency over threaded models have been reported. Our aim has been to improve upon those results reported in [62], which we call "Non-Threaded" models. Furthermore, we aspire in betterment of the modeling paradigm for SDF-like systems.

7. Pure SDF Examples

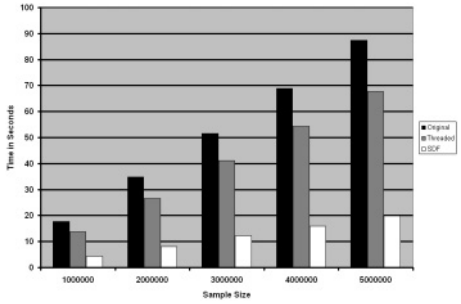
Three systems are modeled using the SDF kernel and the SDF modeling style. They are Finite Impulse Response Filter (FIR), Fast Fourier Transform (FFT), and a Sobel edge detection system. The same systems are modeled with the original standard SystemC DE kernel and the data is shown in Figures 5.6. These experiments are executed on a Linux 2.4.18 platform with an *Intel*® *Pentium*® IV CPU 2.00GHz processor, 512KB cache size, and 512MB of RAM. The first two are taken from SystemC examples distribution.

We already discussed the FIR model in earlier chapters allowing us to move on to the next example, which is the FFT model. The FFT model is also a three-staged model with a Source block, FFT block and a Sink block having the responsibilities of creating the input, performing FFT on the input, and displaying the result, respectively. Figure 5.7 shows the block diagram describing this model.

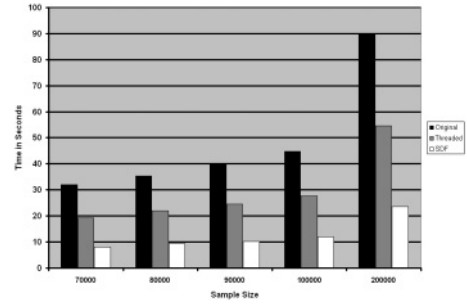
The third example chosen is the Sobel example [53, 62]. This example is a five-staged model shown in Figure 5.8. The Input block simply reads the matrix from an input file and pushes the values into the data path queue. This queue is an input to the CleanEdges block that clears the edges of the matrix and pushes the values through the Channel block to the Sobel operator block. This Sobel block performs the Sobel computations and pushes the results to the Output block completing the entire edge detection example.

We construct these three models with the following underlying scenarios:

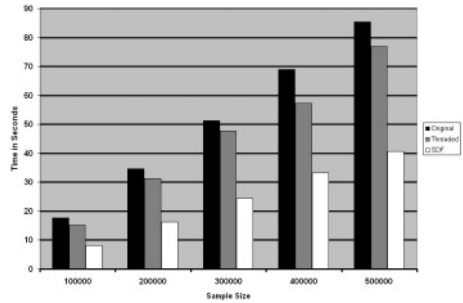
- Discrete-Event kernel using *SC_THREAD()/ SC_CTHREAD()* processes for modeling.



(a) FIR



(b) FFT



(c) Sobel

(Black = Original DE models, Dark Gray = Non-Threaded models, White = SDF models)

Figure 5.6. Results from Experiments

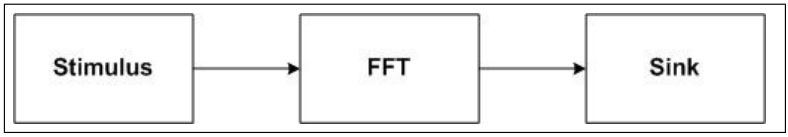


Figure 5.7. FFT Block Diagram

- Discrete-Event kernel with the non-threaded models (transformed using the transformation technique specified in [62]).
- Synchronous Data Flow implementation using the implemented SDF kernel.

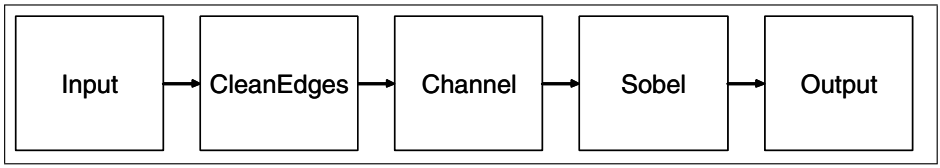


Figure 5.8. Sobel Block Diagram

The graphs in Figure 5.6 present results from these three scenarios. They show that every model demonstrates significant improvement in the amount of simulation time over the original model and the non-threaded model. The three bars on each chart refer to the time taken in seconds for the entire model to be executed in their respective modeling scenarios. The leftmost bar being the original, middle bar being non-threaded are modeled using the DE kernel and the rightmost bar using the SDF kernel. The bar charts show that increasing the number of sample size will still preserve the efficiency presented by the set of collected data. The FIR and FFT yielded approximately 75% improvement in simulation time compared to the original model and the Sobel yielded a 53% improvement in simulation time. Comparing results from the SDF kernel to the non-threaded models, the FIR, FFT and Sobel, showed 70%, 57% and 47% improvement in simulation time respectively. All results show better performance with the SDF kernel than both the original and non-threaded models. We conducted an investigation on the relative lower improvement for the Sobel model and understand that the Input block in Sobel is only executed once to read in the entire matrix of values. However, when this is altered to select segments of the matrix then the performance reflects the FIR and FFT model results. This indicates that simulation efficiency depends on the number of invocations of the blocks required to perform certain functions. The Sobel model using the DE kernel required a lot more invocations when collecting segments of the matrix increasing the signal communication and data passing. Future experimentation proposes comparison with different matrix segment sizes for this edge detection algorithm. The percentage improvement over the non-threaded model in [62] is also consistent with the Sobel edge detection yielding a lower improvement and for the same reason.

Chapter 6

COMMUNICATING SEQUENTIAL PROCESSES KERNEL IN SYSTEMC

Any multi-MoC framework designed to model and simulate embedded systems, or any other complex system composed of concurrently executing components which are communicating intermittently, needs to implement some MoCs that are geared for specific communicating process models. Current SystemC reference implementation lets the user create concurrently executing modules using *SC_THREAD()* or *SC_CTHREAD()* constructs. Modules that have such threads in them communicate via channels which are usually of the *sc_fifo*, *sc_mutex* and other predefined channel types and their derivatives. These channels have blocking and non-blocking read/write interfaces that the threads can call to block themselves or attempt communication with other threads. These thread constructs also can synchronize with clock signals, or events using *wait()* or *wait_until()* function calls directly, or through read/write calls on one of the channel types. Clearly, such threading mechanisms and structures are provided with the Discrete-Event (DE) kernel in mind. What if one wants to model software components which are not necessarily synchronized with a global clock when they suspend or do not need to synchronize with events that are created at the DE kernel level? Often, designers want to model a software system without the notion of clock based synchronization, and later on refine the model to introduce clocks. For such untimed models of concurrent components, designers would much prefer a different MoC than the clock-based DE MoC.

In [28], Communicating Sequential Processes (CSP) is introduced as a model of computation for concurrency that originally dates back to 1978 [27]. In this MoC, sequential processes are combined with process combinators to form a concurrent system of communicating components.

The protocol for communication in such an MoC is fully synchronous as opposed to data flow networks. For example, in data flow networks, buffers in the channels connecting two computing entities are assumed, and, based on buffer size, the computations proceed asynchronous to each other, leaving the communicable data at the buffers for the other components to pick up as and when ready. Of course, in real implementations buffers are of limited size and hence often times requires process blockings. In CSP, the communication happens through a *rendez-vous* mechanism [27]. This necessitates synchronization at the data communication points between the processes, as buffering is not allowed on the channels, and the communicating processes both need to be ready to communicate for communication to take place. If one of the two communicating processes is not ready, the other blocks until both are ready. This imposes a structure and semantics that is amenable to *trace theory*, and in later work to *failure-divergence* semantics. Such theoretical underpinnings make this MoC quite useful for formal analysis, and in recent times formal verification and static analysis tools for CSP models have appeared [40].

CSP provides a convenient MoC for creating a system model which consists of components that need to communicate with each other and their communication is based on synchronous rendez-vous, rather than buffered asynchronous communication. Refining such models with a clocked synchronous model later on is easier than refining a fully asynchronous model. Moreover, the models built can be formally analyzed for deadlock and livelock kind of problems more easily. We therefore picked CSP MoC as one of the first concurrency related MoC for our extension of SystemC.

Rendez-vous Communication

Implementation of the Communicating Sequential Processes Model of Computation requires understanding of rendez-vous communication protocol. Every node or block in a CSP model is a thread-like process that continuously executes unless suspended due to communication. The rendez-vous communication protocol dictates that communication between processes only occurs when both the processes are ready to communicate. If either of the processes is not ready to communicate then it suspends until its corresponding process is ready to communicate, at which it is resumed for the transfer of data.

Figure 6.1 illustrates how the rendez-vous protocol works. T1 and T2 are threads that communicate through the channel labelled C1. T1 and T2 are both runnable and have no specific order in which they are executed. Let us consider process point 1, where T1 attempts to put a value

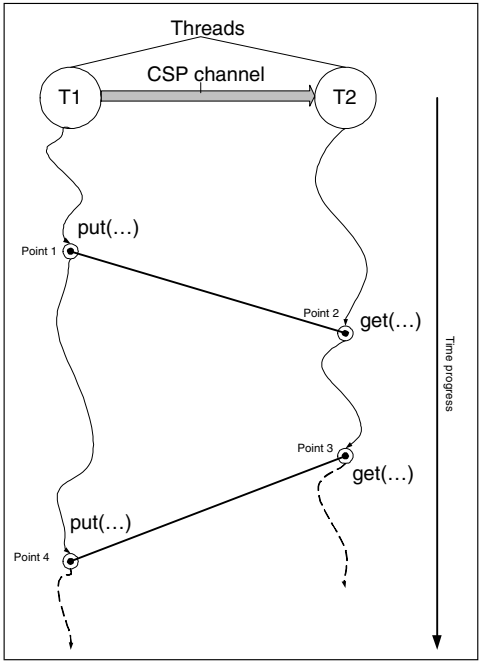


Figure 6.1. CSP Rendez-vous Communication

on the channel C1. However, process T2 is not ready to communicate, causing T1 to suspend when the *put(...)* function within T1 is invoked. When process T2 reaches point 2 where it invokes the *get(...)* function to receive data from C1, T1 is resumed and data is transferred. In this case T2 receives the data once T1 resumes its execution. Similarly, once T2 reaches its second invocation of *get(...)* it suspends itself since T1 is not ready to communicate. When T1 reaches its invocation of *put(...)*, the rendez-vous is established and communication proceeds. CSP channels used to transfer data are unidirectional. That means if the channel is going from T1 to T2, then T1 can only invoke *put(...)* on the channel and T2 can only invoke *get(...)* on the same channel.

1. Implementation Details

We present some design considerations in this section followed by the data structure employed for CSP and implementation details.

1.1 Design Considerations and Issues

Careful thought must be given to the inclusion of a CSP kernel in SystemC. This is necessary because CSP is an MoC disjoint from conventional hardware models. Though CSP is more generally considered

a software MoC, it is an effective MoC when targeting models for concurrency. Clearly, the semantics of CSP are different from the semantics of a Discrete-Event MoC. This implies that, unlike the SDF implementation in SystemC where we targeted for the simulation semantics to remain exactly the same as the DE semantics, in CSP we want them to be completely distinct. Therefore, the Evaluate-Update paradigm is not employed in the implementation.

1.2 Data Structure

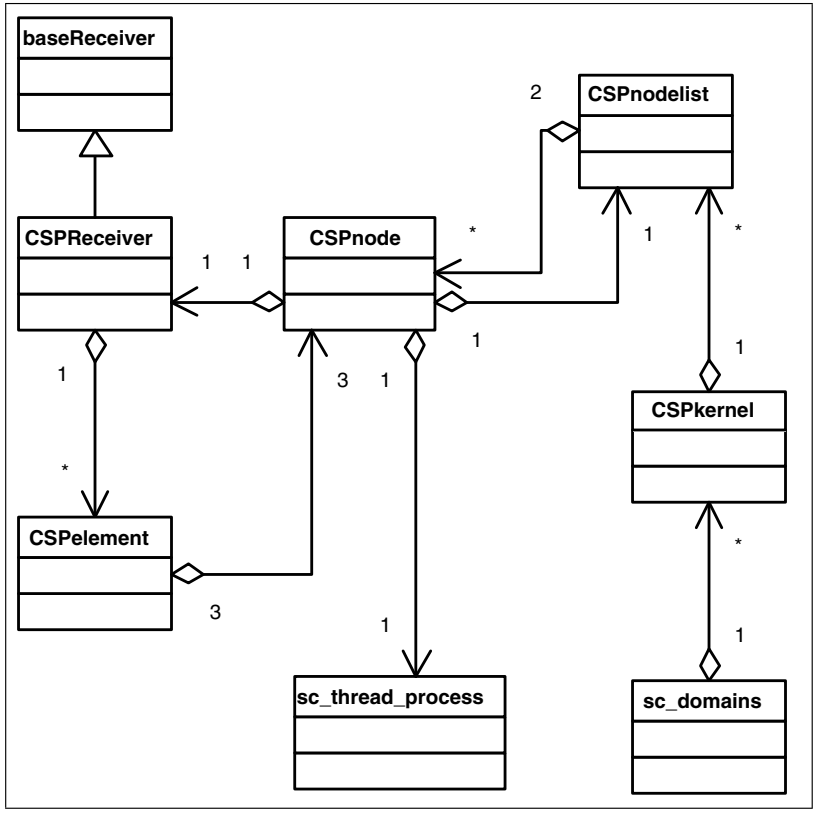


Figure 6.2. CSP Implementation Class Hierarchy

Chapter 4 familiarizes the reader with general implementation class hierarchies that present a minimal organization structure followed by the CSP kernel. This section describes implementation of the class hierarchy shown in Figure 6.2.

A *baseReceiver* class preserves basic information about the receiver that inherits from the *baseReceiver*. This class presently only holds the type of the inheriting receiver, but this can be extended to encompass

Listing 6.1. class baseReceiver

```
1 class baseReceiver  {
2   private:
3     receiverType type;
4
5   protected:
6     receiverType getType();
7     void setType(receiverType t);
8     void setCSP();
9
10  public:
11    baseReceiver();
12    ~baseReceiver();
13
14 };
```

common functionality as described in Chapter 4. Listing 6.1 shows the *baseReceiver* class with an enumerated *receiverType* data type. Variable *type* is set via the derived class, identifying the derived class as a CSP receiver by the use of *setCSP()* function.

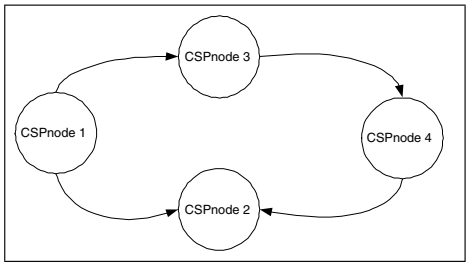


Figure 6.3. Simple CSP model

CSP models require a data structure that represents a graph, which we call a CSP graph (CSPG). The *CSPelement* class is responsible for encapsulating information used to construct this CSPG. Figure 6.3 shows an example of a CSPG. The graph representation is implemented by a list of pointers to objects of type *CSPelement*, which is discussed later in this section. However, for the purpose of creating this CSPG, each object of *CSPelement* contains a pointer to the *CSPnode*s that this *CSPelement* is connected to and from.

From Listing 6.2 *toNode* and *fromNode* point to the objects of type *CSPnode* (defined later in this section) distinguishing the direction of the communication as well. There are two Boolean flags called *putcalled* and *getcalled* that store the state of the channel. The *putcalled* Boolean value is set to *true* if a corresponding *CSPnode* connected to this channel invokes the *put(...)* function call. Similarly, *getcalled* is set when the

Listing 6.2. class CSPelement

```
1 class sc_module;
2 class CSPnode;
3
4 class CSPelement  {
5
6   private:
7     CSPnode * me;
8     CSPnode * toNode;
9     CSPnode * fromNode;
10    static int id;
11    bool putcalled;
12    bool getcalled;
13    csp_event * ev; // store the event that this element is
          going to be triggered on
14
15  public:
16
17    CSPelement();
18    ~CSPelement();
19
20    CSPelement(CSPnode * from, CSPnode * to, int id );
21    CSPelement(CSPnode * from, CSPnode * to );
22    void setid(int i);
23    void setto(CSPnode * to);
24    void setfrom(CSPnode * from);
25    int getid();
26
27    bool getput();
28    bool getget();
29    void setput(bool p);
30    void setget(bool g);
31
32    void setev(csp_event * e);
33    csp_event * getev();
34    void clearev();
35
36    CSPnode* getme();
37    void setme(CSPnode * m);
38
39    CSPnode* getto();
40    CSPnode* getfrom();
41    CSPnode* get_resume_ptr(CSPnode * myself);
42    CSPnode* get_suspend_ptr(CSPnode* myself);
43    string * getmyname(CSPnode * myself);
44
45    //overloaded operators
46    bool operator==(const CSPelement & a) ;
47    bool amIfrom(CSPnode * from);
48
49    friend ostream& operator << (ostream& os, CSPelement & p);
          //output
50 };
```

get(...) function is invoked by its corresponding CSP process. Another Boolean variable typedefed to *csp_event* represents whether there exists an event on the channel. If an event exists then one of the processes connected to this channel was suspended. SystemC events are not used for

csp_event, but regular *bool* data types. This avoids the use of SystemC's DE semantics and events.

Other than general set and get functions for the private members of this class, the important member function is the overloaded *equals* operator. The implementation of this overloaded operator compares the *fromNode* and *toNode* to verify that the *CSPelement* objects on both sides of the *equals* operator have the same addresses for the *fromNode* and *toNode*. If they do, then a particular channel or *CSPelement* that connects two *CSPnode*s is found. The responsibility of *CSPelement* is exactly the same as that of a channel. This is a result of adhering to the general implementation hierarchy, where the CSP channels are effectively represented by *CSPelement* objects. Hence, we inherit *CSPelement* in *CSPchannel*, which is discussed later. This is the mechanism that we employ in searching for the channels through which communication occurs. However, this imposes a limitation that there can only be a maximum of two channels between the two same *CSPnode*s. This gives rise to a problem that if there exists two channels in the same direction between the two same nodes, then according to the equals operator, they will be indistinguishable. Thus, we limit the users to only one channel in the same direction between two *CSPnode*s. We justify this implementation in the following manner:

- By allowing for a templatized data transfer communication that can transfer a data type defined by the user. This allows the user to pass in different values through the communication channel through the user defined data type.

- A single *CSPchannel* can result to multiple suspension points with multiple calls to *get(...)* or *put(...)*.

Figure 6.3 shows a simple CSP model with four CSP processes that are connected via channels. The analogous representation of this simple CSP model using our data structure is shown in Figure 6.4, which shows how objects of *CSPelement* are used to construct a CSPG. The list holding the *CSPelement*s is the *CSPReceiver*. *CSPReceiver* objects are data members of a *CSPnode* that are composed with *CSPelement* objects.

Figure 6.4 shows four *CSPnode*s and their respective *CSPelement*s for the purpose of providing a connection between two CSP processes. The gray box displays objects of *CSPReceiver*. The role of the receiver is simply to encapsulate the *CSPelement*s as shown in Figure 6.4. A simple data structure is employed to represent the CSPG. We employ C++ STL *vector<...>* class to store the addresses of every *CSPelement* inserted in the CSPG and iterate through the list to find the appropriate

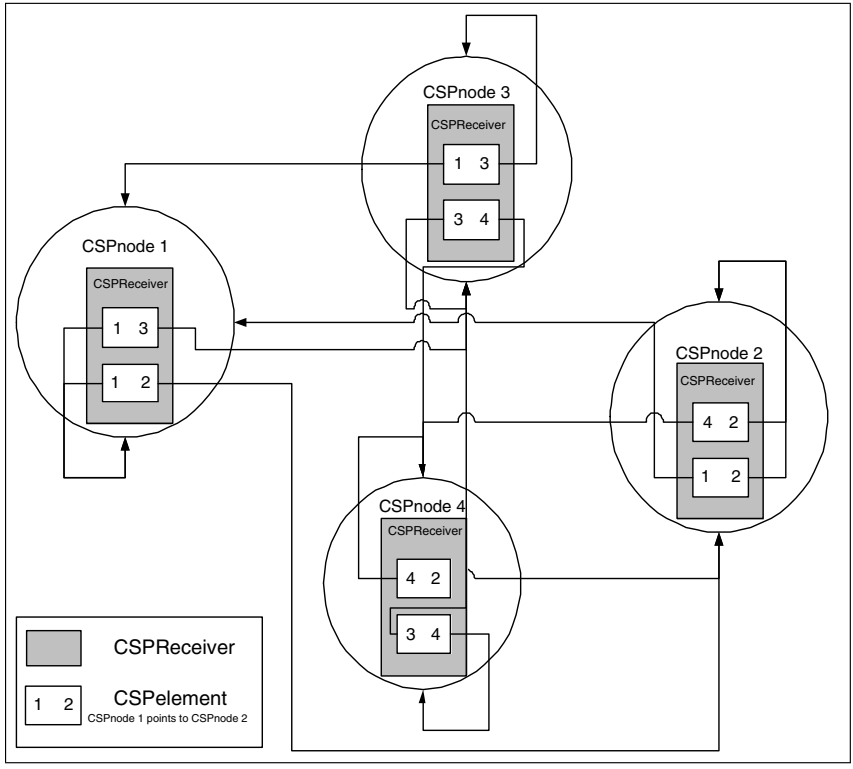


Figure 6.4. Implementation of a Simple CSP Model

channel for communication when required. Every *CSPnode* has its own *CSPReceiver* object that contains the *CSPelement*s that address that particular CSP process. Listing 6.3 displays the class definition describing the *elementlst* as the container of the *CSPelement* addresses along with a private helper function that is used to traverse through the list and identify the requested channel.

We discuss some of the important member functions from this class and their input and output arguments.

put(...):

Inputs:

- A pointer to the *CSPelement* to identify what channel it is to be passed on to.
- The *CSPnode* that is responsible for sending this token.

Outputs:

Listing 6.3. class *CSPReceiver*

```
1 class CSPReceiver: public baseReceiver  {
2
3 private:
4   vector<CSPelement*> elementlst;
5   int id;
6
7   // private helper functions
8   CSPelement* findElement(CSPelement * e);
9
10 public:
11   CSPReceiver();
12   ~CSPReceiver();
13
14   // overloaded Constructors
15   CSPReceiver(CSPnode * fromNode, CSPnode * toNode);
16
17   //functions
18   void get(CSPelement * e, CSPnode * me);
19   void put(CSPelement * e, CSPnode * me);
20
21   void push_into(CSPelement * e);
22   friend ostream& operator<<(ostream& os, CSPReceiver & p) ;
23
24   // event finders
25   csp_event * getevent(CSPelement * el);
26   void setevent(CSPelement * el, csp_event *ev);
27
28   void suspendProc(CSPelement * e, CSPnode * me);
29   void resumeProc(CSPelement * e, CSPnode * me);
30 };
```

- The process suspends if a *get(...)* has not been called on the channel.

get(...):

Inputs:

- A pointer to the *CSPelement* that a token is to be received from.
- The address of the *CSPnode* making the *get(...)* invocation.

Outputs:

- If a *put(...)* has been called the suspended process that called the *put(...)* is scheduled for execution (resumption).

suspendProc(...): Suspends the currently executing thread.

Inputs:

- A pointer to the *CSPelement* that requires suspension due to rendez-vous protocols.

- A pointer to the *CSPnode* that is to be suspended.

Outputs:

- The *CSPnode* currently executing suspends itself.

resumeProc(...): Resumes a particular thread for execution.

Inputs:

- A pointer to the *CSPelement* that is to be resumed due to rendezvous protocols.
- A pointer to the *CSPnode* that is to be resumed.

Outputs:

- The *CSPnode* is scheduled for resumption.

It may seem redundant to supply these functions with the owner of the call, where the owner is the process invoking the member function. However, this is necessary because every *CSPnode* contains all the *CSPelement*s that addresses that process, either as a *fromNode* or a *toNode*. Furthermore, the direction is preserved when inserting the address of the *CSPelement* objects in their respective receiver lists. This is to allow the process to know whether it is the calling process or the called process.

To avoid a convoluted written explanation, let us consider Figure 6.3 where the direction of the communication is from CSPnode 1 and towards CSPnode 3. Our implementation adds a pointer in CSPnode 1's receiver and the same pointer in CSPnode 3's receiver pointing to an object of *CSPelement* whose *fromNode* points to CSPnode 1 and *toNode* is CSPnode 3. For the purpose of the *CSPnode* knowing the direction of communication, it is necessary to compare the process's pointer to both the *fromNode* and *toNode* to realize the direction of communication.

Listing 6.4 defines the *CSPnode* class that encapsulates the *CSPReceiver* as shown in Figure 6.4. Other important private members of this class are *sc_thread* and *my_thread_list*. *sc_thread* holds a pointer to SystemC's *sc_thread_process* object and *my_thread_list* is a pointer to an object that contains a list of *CSPnode*s in a model. These private data members are used during the simulation of the CSP model. The remainder of the member functions are mandatory *set(...)* and *get(...)* functions.

A CSP channel implemented as a class called *CSPchannel* inherits from base class *sc_moc_channel*, but *CSPchannel*s must also support rendez-vous communication as well as the capability to transfer data. For this reason, the *CSPchannel* is specialized. Listing 6.5 shows the definition of this class.

Listing 6.4. class *CSPnode*

```
class CSPnodelist;
class CSPnode {

private:
  CSPReceiver * cspbox; // one CSPnode has one receiver
  int cspid;
  ProcInfo * process;
  sc_thread_handle sc_thread;
  CSPnodelist * my_thread_list;

  // helper functions
  int getid();

public:
  CSPnode();
  ~CSPnode();

  // Set up Process Information
  void setprocaddr(void * a);
  void setprocname(string * n);
  void setprocname(const string & n);
  void* getprocaddr();
  string* getprocname();

  // setup the link between two or more nodes
  void points_to(CSPnode* to);
  void points_to(CSPnode * to, CSPelement * el);
  void points_to(CSPnode & to, CSPelement & el);

  //member functions
  bool send();
  bool send(CSPelement * sendTo);
  void send(CSPelement & sendTo);
  void get(CSPelement * getFrom);
  void get(CSPelement & getFrom);
  bool suspend();
  void portbind(CSPelement *e);

  // set which CSPnodelist it belongs to
  void set_my_thread_list(CSPnodelist * my_list);
  CSPnodelist * get_my_thread_list();

  void print();

  void setnodeev(CSPelement * thisNode, csp_event * e);
  csp_event * getnodeev(CSPelement * getFrom);

  // after execution reschedule immediately
  void reschedule();

  void setmodule(sc_thread_handle mod);
  sc_thread_handle getmodule();
};
```

Notice from Figure 6.5 that multiple inheritance is used to define *CSPchannel*. Inheritance from *sc_moc_channel* and *CSPelement* provides functionality and data structure available in both these base classes. From an object oriented programming sense, the *CSPelement* actually defines a channel between two CSP processes. Thus, the relationships of

Listing 6.5. class *CSPchannel*

```
template <class T> class CSPchannel :
  public CSPelement, public sc_moc_channel<T>
{
  public:

     CSPchannel<T>() {};
     ~CSPchannel<T>() {};

  void push(T & val, CSPnode & node);
  T get(CSPnode & node);
};

template <class T>
  void CSPchannel<T>::push( T & val,CSPnode & node) {
     sc_moc_channel<T>::push(val);
     node.send((CSPelement*) this);

  };

template <class T>
  T CSPchannel<T>::get(CSPnode & node)  {
     node.get((CSPelement*)this);
     return (sc_moc_channel<T>::pop());
  };
```

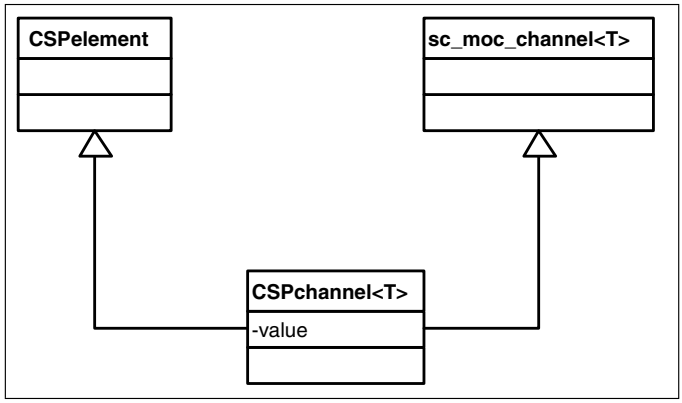


Figure 6.5. Class diagram for *CSPchannel*

CSPchannel is one of an "is a" with both *sc_moc_channel* and *CSPelement*. The member functions in *CSPchannel* are shown in Table 6.1.

Table 6.1. Member function for class *CSPchannel*

Member Function	Purpose
push(...)	Attempts to send a token on the channel
get(...)	Attempts to receive a token from the channel

It follows that there is a need to specialize the *CSPport* class such as to support this specialized *CSPchannel*. Using the *sc_moc_port* base class data structure, *CSPport* appropriately calls member functions of *CSPchannel* when a value is to be inserted or extracted. Listing 6.6 displays the class definition for *CSPport*. The implementation of the *CSPport* class serves the basic purpose of allowing two *CSPnode*s visibility of the *CSPchannel* that connects them. We have implemented overloaded *()* operators to allow CSP port binding. However, we do not perform any checks for port binding errors.

Listing 6.6. class *CSPport*

```
template <class T> class CSPport : public sc_moc_port<T>  {
  public:
    CSPport<T>() {};
    ~CSPport<T>() {};    CSPelement & read();
    void push(T & p, CSPnode & node);
    T get(CSPnode & node);

};

template <class T >
void CSPport<T>::push(T  & p, CSPnode & node)
{
  CSPchannel<T> * castchn = static_cast< CSPchannel<T> * > (port
      );
  if (port != NULL) {
    castchn->push(p, node);
  }
};

template <class T >
T CSPport<T>::get(CSPnode & node)   {
  CSPchannel<T> * castchn = static_cast< CSPchannel<T> * > (port
      );
  return (castchn->get(node));
  };
```

The *CSPnodelist* class shown in Listing 6.7 can be considered to be the class that defines the CSP simulator object in SystemC. Hence, an object of *CSPnodelist* performs the simulation for CSP. The private members are simply two *vector<...>* lists where *nodelist* is the list of pointers to all the *CSPnode*s and *runlist* is a list of the runnable CSP processes. Though the *runlist* is of type *vector<...>* we have implemented a queue with it. This behavior is necessary to correctly simulate a CSP model. Other private members are pointers to the coroutine packages used to implement QuickThreads [35] in SystemC. *m_cor* identifies the executing simulation context's coroutine whereas *m_cor_pkg* is a pointer to a file static instance of the coroutine package through which blocking and resumption of thread processes can be performed. For further details about QuickThread implementation in SystemC please refer to Appendix A.

Coroutine is SystemC's implementation of the QuickThread core package as the client package.

Some of the important member functions are listed below:

void push_runnable(CSPnode & c) The *CSPnode* is pushed onto the *runlist* such that it can be executed.

CSPnode * pop_runnable() Retrieves the top runnable thread.

void next_thread() Selects the next CSP process to execute.

void sc_csp_switch_thread(CSPnode * c) Used in blocking the currently executing thread and resuming execution of the thread identified by the pointer *c*.

sc_cor* next_cor() Retrieves a pointer to the next thread coroutine to be executed.

Implementation details of these classes are not presented, but we direct the reader to refer to implementation details available at our website [36]. This brief introduction of the CSP data structure allows us to proceed to describing how the CSP scheduling and simulation is performed. For some readers it may be necessary to refer to Appendix A where we describe the coroutine package for SystemC based on [35].

2. CSP Scheduling and Simulation

Table 6.2. Few Important Member Functions of CSP Simulation class *CSPnodelist*

Method / Variable name	Maintained by	
	CSP Kernel	QuickThread Package
runlist	✓	-
nodelist	✓	-
m_cor_pkg	✓	✓
m_cor	✓	✓
push(...)	✓	-
push_runnable(...)	✓	-
sc_switch_switch_thread(...)	✓	-
pop_runnable(...)	✓	-
next_cor(...)	✓	-
run_csp(...)	✓	-

Simulation of a CSP model uses a simple queue based data structure that contains pointers to all the *CSPnodes*. This queue is constructed by using C macros that work similar to the existing *SC_THREAD()* macros. We introduce the macro *SC_CSP_THREAD()* that takes three

Listing 6.7. class CSPnodelist

```
1 class CSPnodelist {
2
3   private:
4     vector<CSPnode* > * runlist;
5     vector<CSPnode* > * nodelist;
6
7   public:
8     CSPnodelist();
9     ~CSPnodelist();
10
11    void push_runnable(CSPnode & c);
12    void push(CSPnode & c);
13
14    void next_thread();
15
16    CSPnode * pop_runnable();
17    void removefront();
18
19    //sizes of lists
20    int nodelist_size();
21    int runnable_size();
22
23    void csp_trigger();
24    void runcsp(CSPnodelist & c);
25     sc_cor_pkg* cor_pkg()
26        { return m_cor_pkg; }
27    sc_cor* next_cor();
28
29    vector<CSPnode*> * getnodelist();
30    vector<CSPnode*> * getrunlist();
31    void init();
32    void clean();
33    void initialize(bool nocrunch);
34
35    void sc_csp_switch_thread(CSPnode * c );
36    void print_runlist();
37
38    void push_top_runnable(CSPnode & node);
39
40  private:
41    sc_cor_pkg*                m_cor_pkg; // the simcontext's
          coroutine package
42    sc_cor*                    m_cor;     // the simcontext's
          coroutine
43
44 };
```

arguments: the entry function, the *CSPnode* object specific for that *SC_CSP_THREAD()* and the *CSPnodelist* to which it will be added. This macro calls a helper function that registers this CSP thread process by inserting it in the *CSPnodelist* that is passed as an argument.

Invoking the function *runcsp(...)*, initializes the coroutine package and the current simulation context is stored in the variable *main_cor*. The simulation of the CSP model starts by calling the *sc_csp_start(...)* function. Table 6.2 shows a listing of some important functions and variables and whether the CSP kernel or the QuickThread package manages

them. The variable *m_cor_pkg* is a pointer to the file static instance of the coroutine package. This interface for the coroutine package is better explained in Appendix A. All thread processes require being prepared for simulation. The role of this preparation is to allocate every thread its own stack space as required by the QuickThread package. After this preparation, the first process is popped from the top of the *runlist* using *pop_runnable(...)* and executed. The thread continues to execute until it is either blocked by executing another thread process or it terminates. This continues until there are no more processes on the *runlist*.

Listing 6.8. class csp_trigger() function

```
1 void CSPnodelist::csp_trigger() {
2   while (true)  {
3       sc_thread_handle thread_h = pop_runnable()->getmodule();
4       removefront();
5       while( thread_h != 0 && ! thread_h->ready_to_run() ) {
6         thread_h = pop_runnable()->getmodule();
7         removefront();
8       }
9       if( thread_h != 0 ) {
10        m_cor_pkg->yield( thread_h->m_cor );
11      }
12
13      if( runnable_size() == 0 ) {
14        // no more runnable processes
15        break;
16      }
17  };
18}
```

We present the function *csp_trigger()* in Listing 6.8 that is responsible for performing the simulation. The *pop_runnable()* function extracts the topmost pointer to a *CSPnode* that has an *sc_thread_handle* as a private member, which is retrieved by invoking the *getmodule()* member function. The *m_cor_pkg→yield(thread_h→m_cor)* function invokes a function implemented in the *sc_cor_qt* class. This *yield(...)* function is responsible for calling a helper function to switch out the currently executing process, saving it on its own stack and introducing the new process for execution. The process coroutine is sent by the *thread→m_cor* argument. A check is done if the runnable queue is empty and then the simulation is terminated. However, most CSP processes are suspended during their execution, which requires brief understanding of how blocking is performed using QuickThreads. For most readers it will suffice to explain that when a process suspends via the *suspendProc(...)* function, the state of the current process is saved and a helper function called *next_cor()* is invoked. The *next_cor()* returns a pointer of type *sc_cor* which is the coroutine for the next thread to execute.

Listing 6.9. function next_cor() function

```
1 sc_cor* CSPnodelist::next_cor()
2 {
3   sc_thread_handle thread_h = pop_runnable()->getmodule();
4   removefront();
5   while( thread_h != 0 && ! thread_h->ready_to_run() ) {
6     thread_h = pop_runnable()->getmodule();
7     removefront();
8   }
9   if( thread_h != 0 ) {
10    return ( thread_h->m_cor );
11  } else
12  return m_cor;
13 }
```

Implementation of the *next_cor()* function is similar to the *csp_trigger()* function. This is because once a CSP process is suspended, the next process must continue to execute. So, *next_cor()* implements a similar functionality as *csp_trigger()* with the exception of calling *yield(...)* on the process to execute, and the coroutine is returned instead. Furthermore, if there are no more processes on the *runlist*, then the main coroutine of the simulation is returned by returning *m_cor* as shown in Listing A.11. Therefore, the suspension of processes is in essence performed by yielding to another process, where QuickThreads serve their purpose by making it relatively simple for blocking of thread processes. Likewise, resumption of the threads is simple as well. Using the coroutine package, resumption is done by rescheduling the process for execution. Therefore, when *resumeProc(...)* is invoked, the address of the process to be resumed is inserted into the *runlist* queue. Once the top of the queue reaches this process, the thread is resumed for execution. During modeling, non-deterministic behavior is introduced by randomization in the user constructed models. According to this implementation, CSP models have the potential for executing infinitely such as the Dining Philosopher problem. We visit the implementation of this example using our CSP kernel for SystemC.

3. Example of CSP Model in SystemC

Early in Chapter 2, we introduced the Dining Philosopher problem. A schematic of the way it can be implemented is shown in Figure 6.6. In this section, we revisit this example and provide the reader with modeling guidelines along with code fragments to describe how it is modeled using our kernel. However, during our earlier discussion, we did not present the possibility of deadlock. A deadlock occurs in the Dining Philosopher problem when for instance every philosopher feels hungry

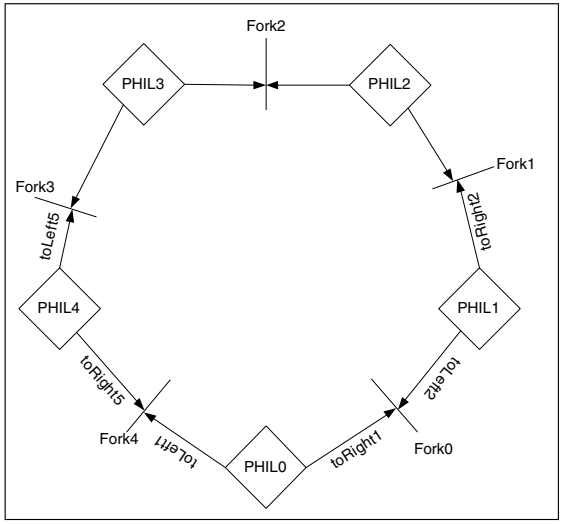


Figure 6.6. CSP Implementation of Dining Philosopher

and picks up the fork to their left. That prevents any of the philosophers eating since two forks are required to eat causing the model to deadlock. We use a simple deadlock avoidance technique where we have a footman that takes the philosophers to their respective seats and, if there are four philosophers at the table, asks the fifth philosopher to wait and seats him only after one is done eating. This is a rudimentary solution, but for our purpose it is sufficient.

We begin by describing the module declaration of a philosopher in Listing 6.10. The original implementation that we borrow is available at [60]. That implementation is a pure C++ based implementation that we modify to make a CSP SystemC example. Each philosopher has a unique *id* and an object of *ProcInfo*. This *ProcInfo* class is implemented as a debug class to hold the address of the process and the name of the process purely for the reasons of output and debugging. The full source description will have the implementation of this class, though we do not describe it since it is not directly relevant to the implementation of the CSP kernel in SystemC. There is an instantiation of a *CSPnode* called *csp* through which we enable our member function invocations for CSP and two *CSPport*s, *toRight* and *toLeft*. The *toRight* connects to the *CSPchannel* that connects the philosopher to the fork on its right and *toLeft* to the one on its left. There are several intermediate functions defined in this module along with the main entry function. The entry function is called *soln()* that is bound to a CSP process through the *SC_CSP_THREAD()* macro.

Listing 6.10. Philosopher Module Declaration

```
SC_MODULE(PHIL) {

  int id;
  int st;
  string strid;
  int    _timeToLive;

  CSPnode csp;
  CSPport<int> toRight;
  CSPport<int> toLeft;

  int * drop;
  int * pick;
  ProcInfo proc;

  void askSeat(int id);
  void getfork();
  void dropfork();
  void soln();
  int getstate();
  void print();

  // footman required for deadlock free solution
  bool reqSeat();

  SC_CTOR(PHIL) {
     st = -1;
     SC_CSP_THREAD(soln, DP, csp)  {
     };
  };
};
```

We begin describing the implementation of the *PHIL* class by displaying the entry function *soln()* as shown in Listing 6.11. Many print statements are inserted to view the status of each of the philosophers and the forks. This is handled by the *print_states()* function. However, the core functionality of the entry function begins by invoking *getfork()*. Listing 6.12 shows the implementation of this function. The *state[x]* array is global and simply holds the state value for every philosopher, which is updated immediately to allow the *print_states()* to output the updated values. The philosopher requests a fork on either the left or right of himself by calling the *get(...)* member function on the port. If the fork is available to be picked up and has been recognized by the *CSPchannel* then the philosopher process will continue execution and request the other fork. However, if the fork is not ready to be picked up, this process will suspend.

Once the philosopher has both the forks in hand, *soln()* goes to its eating state where we simply output *EATING* and wait for a random amount of time defined by functions from [60]. After the eating state,

Listing 6.11. function soln()

```
1 void PHIL::soln() {
2     int duration = _timeToLive;
3     int eatCount = 0;
4     int totalHungryTime = 0;
5     int becameHungryTime;
6     int startTime = msecond();
7
8     while(1) { // ( msecond() - startTime < duration * 1000 ) {
9
10      if ((reqSeat() == true) && ((state[id] != 0) || (state[id
            ] != 6))) {
11        becameHungryTime = msecond();
12        print_states();
13        cout << " PICKING UP FORKS " << endl;
14        getfork();
15        cout << " DONE PICKING UP FORKS " << endl;
16        print_states();
17        totalHungryTime += ( msecond() - becameHungryTime );
18        eatCount++;
19        cout << " EATING " endl;
20        state[id] = 3;
21        usleep( 1000L * random_int( MeanEatTime ) );
22        cout << " DONE EATING " << endl;
23        print_states();
24        cout << " DROPPING FORKS " << endl;
25        dropfork();
26        usleep( 1000L * random_int( MeanThinkTime ) );
27        cout << " DONE DROPPING FORKS " << endl;
28        print_states();
29        cout << " THINKING " << endl;
30        state[id] = 6;
31        usleep( 1000L * random_int( MeanThinkTime ) );
32        state[id] = 0;
33        print_states();
34        --space;
35        csp.reschedule();
36      } else {
37          cout << " STANDING " << endl;
38          csp.reschedule();
39      }
40    }
41    state[id] = 7;
42    totalNumberOfMealsServed += eatCount;
43    totalTimeSpentWaiting += ( totalHungryTime / 1000.0 );
44    cout << "Total meals served = " << totalNumberOfMealsServed
          << "\n";
45    cout << "Average hungry time = " <<
46    ( totalTimeSpentWaiting / totalNumberOfMealsServed ) << "\n"
          ;
47 };
```

the philosopher enters the state where he attempts to drop the forks by calling *dropfork()* described in Listing 6.13.

Dropping of the forks is modeled by sending a value on the channel which is performed via the *push(...)* on the port. If the *push(...)* is invoked without the corresponding CSP node at the end of the channel ready to accept the token, the process will suspend. Returning back to the entry function, after the forks have been dropped there is a random

Listing 6.12. function getfork()

```
1 void PHIL::getfork()  {
2   if ( numPhil % 2 )  {
3     // even-numbered philosophers pick left then right
4     state[id] = 1;
5     print_states();
6     toLeft.get(csp);
7
8     state[id] = 2;
9     print_states();
10    toRight.get(csp);
11  }
12  else  {
13    // odd-numbered philosopher; pick right then left
14    state[id] = 2;
15    print_states();
16    toRight.get(csp);
17
18    state[id] = 1;
19    toLeft.get(csp);
20    print_states();
21    }
22 };
```

Listing 6.13. function dropfork()

```
1 void PHIL::dropfork() {
2   // drop left first, then right not that it matters
3   state[id] = 4;
4   print_states();
5   toLeft.push(*drop, csp);
6   state[id] = 5;
7   print_states();
8   toRight.push(*drop, csp);
9 };
```

usleep(...) that suspends execution for microsecond intervals. This completes the eating process for the philosopher such that he returns to his thinking state followed by a random valued *usleep(...)*. According to the queue based implementation, once the process completes its first iteration of the entry function, it must be rescheduled so that the process address is added onto the *runlist*. We provide the *reschedule()* function that the user must invoke to reinsert the CSP process address into the *runlist*.

For the behavior of the fork, we define the module as shown in Listing 6.14. The *FORK* module also has an *id* to differentiate the different forks on the table, an integer valued variable *queryFork* that represents the state of the fork where 1 means that the fork is down and -1 means the fork is not down. There is an instance of a *CSPnode* object called *csp* and two *CSPports* called *fromRight* and *fromLeft*. The *fromLeft* port connects to a *CSPchannel* coming from the *toRight* port of a philosopher

Listing 6.14. Module FORK

```
SC_MODULE(FORK) {
  int id;
  int queryFork;
  CSPnode csp;
  CSPport<int> fromRight;
  CSPport<int> fromLeft;
  int * drop;
  int * pick;

  ProcInfo proc;

  void reqFork();
  void addressFork();

  SC_CTOR(FORK) {
    queryFork = 1;
    SC_CSP_THREAD(addressFork, DP, csp);
  };
};
```

and the *fromRight* connects to the neighboring philosopher's *toLeft*. The entry function *addressfork()* is described next.

The *addressFork()* function dictates the fork's behavior. This behavior is dependent on the state of the philosophers. Listing 6.15 shows that there are four cases that have implementation for the fork. Cases 1 and 2 only occur when the fork is down on the table and cases 4 and 5 only occur when the fork is not available on the table. We implemented a function that gets the *id*s of the philosophers that surround the fork. We use simple tricks with the *id* of the forks and philosophers to locate the neighbors as shown in Listing 6.16. Our heuristic for finding the neighbors involves looking to the left of the fork and then the right of the fork. We identify each fork with a corresponding *id* as well. Based on this *id* we locate the *id*s of the neighboring philosophers with sufficient cases to ensure that *id*s of forks with *id* 4 and 0 perform an appropriate wrap around to complete the circular setup as shown in Figure 6.6. The *addressFork()* function checks the state of the neighbors and accordingly either gives itself (the fork) to the philosopher or requests itself back, otherwise it simply does nothing. We list the functionality of the fork as follows:

Case 1: The philosopher to the right has requested a fork so the fork gives itself through the *put(...)* function to the philosopher on the right.

Case 2: The philosopher to the left has requested this fork, so the fork gives itself to the requesting philosopher since the fork is still down for Cases 1 and 2.

Listing 6.15. addressfork() member function

```
void FORK::addressFork()  {
  while(true) {
  // Get my neighbors
  int * nbors = get_my_neighbors(id);
  bool resched = false;
    for (int i =0; i < 2; i++)  {
      cout << "PHIL_" << nbors[i]+1 << " FORK_" << id+1 ;
      switch(state[nbors[i]]) {
        case 1: {
          // Guy asks on his Left so Send Right
          if ((i != 0) && (queryFork ==1))  {
            queryFork = -1;
            forks[id] = queryFork;
            state[nbors[i]] = 8;
            print_states();
            fromRight.push(*pick, csp);
          }
          break;
        }
        case 2: {
          // Guy asks on his Right so Send Left
          if ((i != 1)  && (queryFork ==1)){
            queryFork = -1;
            forks[id] = queryFork;
            state[nbors[i]] = 9;
            print_states();
            fromLeft.push(*pick, csp);
          }
          break;
        }
        case 4: {
          if ((i != 0)  && (queryFork !=1)){
            queryFork = 1;
            forks[id] = queryFork;
            print_states();
            fromRight.get(csp);
          }
          break;
        }
        case 5: {
          if ((i != 1)&& (queryFork !=1)) {
            queryFork = 1;
            forks[id] = queryFork;
            print_states();
            fromLeft.get(csp);
          }
          break;
        }
        default:  {
          break;
        };
      };
      cout << "\t";
      };
      csp.reschedule();
    delete nbors;
  } // END WHILE

};
```

Listing 6.16. get_my_neighbors function

```
1 int * get_my_neighbors(int id)  {
2   int * nbors = new int[2];
3
4   if ((id != 0) && (id != 4)) {
5     nbors[0] = id ;
6     nbors[1] = id +1;
7   } else  {
8     if (id == 0)  {
9       nbors[0] = 0;
10      nbors[1] = id + 1;
11
12    }else {
13      if (id == 4)  {
14        nbors[0] = id;
15        nbors[1] = 0;
16      };
17    }
18  }
19  cout << "----==== PHIL_" << nbors[0]+1 << "  FORK_" << id
        + 1 << "    PHIL_" << nbors[1]+1 << "  ====----" << endl;
20
21  return nbors;
22 };
```

Case 4: The fork was given to the philosopher on the right so request the fork back from the philosopher.

Case 5: The fork was given to the philosopher on the left so this is requested back.

This model of the Dining Philosopher executes infinitely unless the conditions are un-commented in the *soln()* function [Listing 6.11, Line 8] which causes the *while()* loop to execute for a limited number of executions and terminates, causing the philosophers to in essence, die (perhaps die from over eating).

4. Modeling Guidelines for CSP Models in SystemC

There are some basic modeling guidelines that the implementation of the CSP kernel in SystemC imposes. A modeler should follow a particular scheme in constructing such models. To better understand these construction rules we present some basic modeling guidelines as follows:

1 Only use *CSPchannel*s for unidirectional communication as per CSP specifications.

2 Every *SC_MODULE()* can have multiple CSP processes initialized as long as there is no multiplicity in the communication channels between the same two CSP processes.

3 The current version of the CSP kernel requires instantiation of a *CSPnodelist* that is accessible by all modules so the use of the keyword *extern* may be required if separate files are used for creating models.

4 The simulation can be initialized by calling the member function *runcsp(...)* of the *CSPnodelist* object.

5 Simulation begins by invoking *sc_csp_start(...)*.

6 It may be necessary to update global variables such as the *state[x]* array in the Dining Philosopher problem to allow interpretation of immediate behaviors and responses.

7 Non-deterministic behavior may require the use of randomization functions.

5. Example of Producer/Consumer

A trivial example using CSP is the Producer/Consumer model. This model is simple and has two processes, a Producer, a Consumer and one channel between them. The communication direction between the processes goes from the Producer to the Consumer. This example is similar to the *simple_fifo* example in the SystemC distribution. The differences are that the processes are CSP processes and instead of an *sc_fifo* channel between the processes, there is a *CSPchannel*.

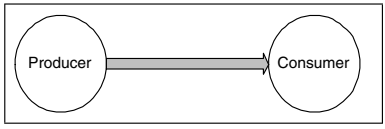


Figure 6.7. Producer/Consumer Example in CSP

Listing 6.17 shows the module declaration for the *PRODUCER* class. Notice an instance of *CSPnode* and a *CSPport*. The *production* pointer holds the string that the Producer sends to the Consumer one character at a time [Listing 6.17, Line 5]. In [Listing 6.17, Line 12], the *at(...)* member function from the *string* class returns a character at the location defined by the argument and stores it in a variable *ch*. This character is pushed onto the channel by invoking the *push(...)* member

function on the port that connects the two CSP processes. An instance of *CSPnodelist* labelled as *DP* is accessible by both the *PRODUCER* and *CONSUMER* objects.

The *if* construct repeatedly sends the same string by the Producer when the *sz* string location counter is equal to the number of characters in the string. This makes the model run infinitely. The constructor of *PRODUCER* module sets the *production* pointer to a string and invokes the *SC_CSP_THREAD()* macro for registering this process as a CSP process.

Listing 6.17. PRODUCER module declaration

```
SC_MODULE(PRODUCER) {

  CSPnode csp;
  CSPport<char> toConsumer;
  string * production;
  ProcInfo proc;

  void sendChar() {
    int sz = 0;
    while(1)
    {
      char ch = production->at(sz);
      ++sz;
      toConsumer.push(ch, csp);
      //csp.send((token)&ch, toConsumer.read());
      // allow for infinite execution
      if (sz == (signed) production->size())
        sz = 0;
      csp.reschedule();
    }
  };

  SC_CTOR(PRODUCER) {
    production = new string();
    *production = "This is a test string for Produced/Consumer
        example :]";
    SC_CSP_THREAD(sendChar, DP, csp);
  };
};
```

The Consumer process shown in Listing 6.18 again has an instance of *CSPnode* and *CSPport*. The Consumer accepts a character from the channel and prints it out. The constructor is straightforward where *SC_CSP_THREAD()* macro registers the *CONSUMER* class as a CSP process.

The driver program for this model is presented in Listing 6.19. The channel that connects the Producer and Consumer is *ptoc*. This channel is bound with the processes' respective ports. The direction of the channel is set by using the *points_to(...)* member function from the *CSPnode* class. *runcsp(...)* prepares the CSP simulation for execution, and a global function *sc_csp_start(...)* triggers this CSP model.

Listing 6.18. *CONSUMER* module declaration

```
1 SC_MODULE(CONSUMER) {
2   CSPnode csp; // CSP node
3   CSPport<char> fromProducer;
4   ProcInfo proc;
5
6   void getChar()  {
7     while (1)
8     {
9       char ch ;
10      ch = fromProducer.get(csp);//(char*)csp.get(fromProducer.
            read()) ;
11      cout << "<<<<<<<<<<< Received " << ch << endl;
12      csp.reschedule();
13    }
14  };
15
16  SC_CTOR(CONSUMER) {
17    SC_CSP_THREAD(getChar, DP, csp);
18  };
19 };
```

Listing 6.19. Driver program for Producer/Consumer Example

```
1 int sc_main(int argc, char *argv[]) {
2   CSPchannel<char> ptoc; // Channel from Producer to Consumer
3   PRODUCER p("Producer"); // Producer Instance
4   p.toConsumer(ptoc); // Bind Producer
5   p.csp.setprocname("Producer"); //Debug information
6
7   CONSUMER c("Consumer"); // Consumer Instance
8   c.fromProducer(ptoc); // Bind Consumer
9   c.csp.setprocname("Consumer"); //Debug information
10
11  p.csp.points_to(c.csp, ptoc); // Set direction of channel
12
13  DP.runcsp(DP); // Prepare CSP for execution
14  sc_csp_start("",&DP); // Start simulation
15  return 0;
16 };
```

6. Integrating CSP & DE kernels

An understanding of QuickThreads and their implementation in SystemC as coroutine packages is required for integrating these two MoCs. We advise the reader to read Appendix A for a better understanding of QuickThreads and coroutine packages in SystemC.

Appendix A explains the workings of the reference kernel for the SystemC scheduler with focus on the coroutine packages. We briefly reiterate how SystemC manages its coroutines and thread processes. Our interest is primarily in thread processes because CSP processes are also thread processes that we want to schedule differently and separate from the DE kernel. The SystemC scheduler initializes thread processes by

creating stack space along with initializing the stack with the appropriate function and its arguments. After thread initialization, the threads are executed by invoking the *yield(...)* function from the *sc_cor_pkg* that switches out the current executing process and prepares the new process (passed via the argument of the function) to execute. Suspension functions such as *wait(...)* perform this switch to allow other runnable processes to execute. The QuickThread package uses *preswitch* for context switching that allows for this implementation. A function called *next_cor(...)* is used to determine the next thread to execute. Once the runnable queues are empty, the control is returned to the main coroutine identified by the *main_cor* coroutine. This main coroutine can also be suspended, which is what happens when a new thread process is scheduled for execution. It is also resumed after no more thread processes are *runnable*.

Listing 6.20. Overloaded Constructor and helper function in *sc_cor_pkg_qt*

```
1 sc_cor_pkg_qt::sc_cor_pkg_qt( CSPnodelist* simc )
2 : sc_cor_pkg( simc )
3 {
4   if( ++ instance_count == 1 ) {
5     // initialize the current coroutine
6     assert( curr_cor == 0 );
7     curr_cor = &main_cor;
8
9 }
10
11 sc_cor*
12 sc_cor_pkg_qt::get_demain()
13 {
14     return curr_cor;
15 }
```

Different semantics for Discrete-Event based simulation and CSP simulation justifies the need for separation of these two kernels. However, SystemC reference implementation treats the *sc_simcontext* class as the toplevel scheduler class with the main coroutine and coroutine package accessible only through an instance of *sc_simcontext*. For isolation, we included functionality in the CSP encapsulation to have pointers to the coroutine package and the main coroutine. We also implemented a CSP-specific *next_cor()* function along with several other thread core functions discussed earlier. The CSP kernel as a stand-alone kernel works without any concerns. However, we encounter an interesting problem when invoking the DE kernel to execute a DE model. As we know, SystemC is designed as a single scheduler simulation framework, which means the coroutine package is created from the *sc_simcontext* class in the *initialize(...)* function. When trying to invoke *initialize(...)* while in a CSP

simulation, the loss of process stack space is experienced. This is due to a singleton pattern used in creating SystemC's DE scheduler. Hence, only one instance of the coroutine package must exist and given that we attempt to invoke the DE kernel from within the CSP kernel, the DE kernel must address the coroutine package created in the CSP kernel instance. This requires a couple of changes in the the coroutine package files and the *sc_simcontext* class. We first discuss the changes we made in the coroutine packages.

Listing 6.21. Overloaded Constructor in *sc_cor_pkg* class

```
1 class sc_cor_pkg
2 {
3 public:
4     ...
5     // overloaded constructor
6     sc_cor_pkg( CSPnodelist* simc )
7         : m_simcsp( simc ) { assert( simc != 0 ); }
8     ...
9
10    void setsimc(sc_simcontext * simc) { m_simc = simc; };
11    void set_csp(CSPnodelist * csp) { m_simcsp = csp; };
12
13    // get the simulation context
14    sc_simcontext* simcontext()
15        { return m_simc; }
16    CSPnodelist * cspcontext()
17        { return m_simcsp;}
18 private:
19
20    sc_simcontext* m_simc;
21    CSPnodelist * m_simcsp;
22 private:
23    ...
24 };
```

Creating an instance of type *sc_cor_pkg_qt* makes a check for having one instance with the *instance_count* and its interface class constructor is also invoked. An object of *sc_cor_pkg_qt* results in the constructor of *sc_cor_pkg* being invoked. Hence, the overloaded constructor described in Listing 6.20 invokes the constructor of class *sc_cor_pkg* with an argument containing the *CSPnodelist* pointer. A helper function *get_demain()* is added to retrieve the *curr_cor* that signifies the current executing context. We use this to make a call-back to the process that performs the invocation of the DE kernel. The interface also undergoes modification to accommodate calls to the interface to extract the correct information. Listing 6.21 displays the additions to the *sc_cor_pkg* class.

A pointer to the *CSPnodelist* is added as a private variable and its respective member functions to set and get address of this pointer. These are the changes that have to be done in the coroutine packages to allow for a CSP model to execute using the coroutine package. At this point we

Listing 6.22. *next_cor()* member function in class *sc_simcontext*

```
1 sc_cor* sc_simcontext::next_cor()
2 {
3     if( m_error ) {
4       return m_cor;
5     }
6     sc_thread_handle thread_h = pop_runnable_thread();
7     while( thread_h != 0 && ! thread_h->ready_to_run() ) {
8       thread_h = pop_runnable_thread();
9     }
10    if( thread_h != 0 ) {
11      return thread_h->m_cor;
12    } else {
13        return (oldcontext);
14    }
15 }
```

only show the inclusion of one *CSPnodelist* (one CSP model) addressed by the coroutine packages. However, we plan to extend this later to support multiple CSP models using the same coroutine package.

We are considering invocations of the DE kernel through the CSP kernel, which requires altering the initialization code for the *sc_simcontext* class. We need to point the *m_cor_pkg* private member of class *sc_simcontext* to the *sc_cor_pkg* pointer in the *CSPnodelist* class. This is performed by invoking the *cor_pkg()* from the *CSPnodelist* followed by an invocation of *get_main()* to retrieve the main coroutine. We introduce a new private data member in *sc_simcontext* called *oldcontext* of type *sc_cor**, which we set by invoking the *get_demain()* member function on variable *m_cor_pkg*. We use *oldcontext* during the *next_cor()* function for class *sc_simcontext* as shown in Listing 6.22.

Variable *oldcontext* is returned when there are no more runnable threads in the system, similar to the original implementation of the *next_cor()* function where *main_cor* was being returned. The purpose of saving *oldcontext* is to allow the simulation to return to the coroutine that invoked the DE kernel. Suppose a CSP process invokes a DE kernel for some computation. *oldcontext* would then store the coroutine of the calling CSP process. The DE simulation returns to *oldcontext* once it has no more processes for execution, resuming the execution of the calling CSP process.

We illustrate the invocation of the DE kernel from the CSP in Figure 6.8. The assigned addresses are made up and do not resemble real addresses in our simulation, but we merely present them to further clarify the manner in which the *oldcontext* is used. During initialization of the CSP model, shown by the CSP block, *m_cor_pkg* and *main_cor* are set to their correct addresses. Every thread process has an *m_cor* variable

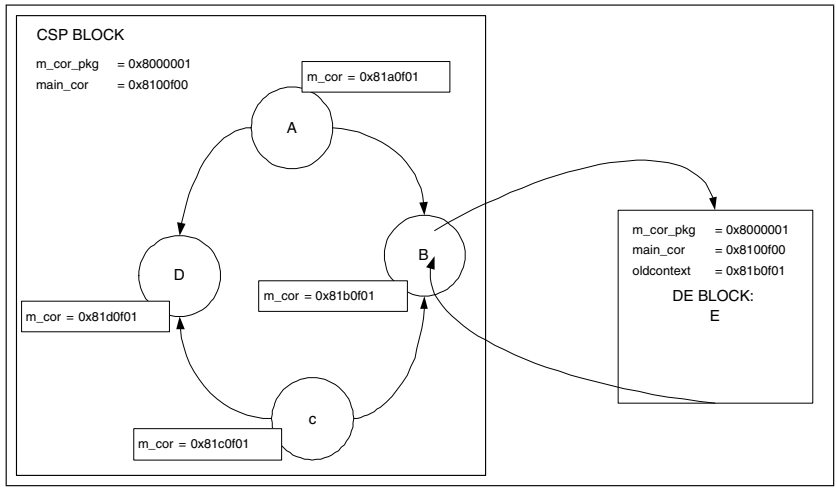


Figure 6.8. Example of DE kernel invocation in CSP

that holds the coroutine for that particular thread. At some point during the execution of process B, a DE model is supposed to execute. This DE model requires that the CSP kernel yield to the DE kernel to simulate the DE block. Hence, the initialization functions of the DE kernel are called where the addresses of the private data members *m_cor_pkg* and *main_cor* are extracted from the CSP kernel and the current simulation context is saved in *oldcontext*. Notice that the address of the *oldcontext* is the same as the *m_cor* value of process B. According to the *next_cor(...)* function definition in Listing 6.22, *oldcontext* is returned once there are no more threads to execute, implying that once the DE simulation model is complete and there are events to be updated, the scheduler returns control to *oldcontext* which is the calling CSP thread. This in effect allows for DE kernel invocations from CSP as we show via an implemented example in Chapter 9.

Chapter 7

FINITE STATE MACHINE KERNEL IN SYSTEMC

Constructing a Finite State Machine (FSM) model in SystemC is possible with current modeling constructs of SystemC. This means that the existing SystemC can effectively provide means of constructing an FSM model. Some may argue that given a Discrete-Event simulation kernel, there is no need to add a Finite State Machine (FSM) kernel for SystemC. However, with the vision of a truly heterogeneous modeling environment in SystemC, the need for such an inclusion is arguable. Furthermore, with hierarchy in mind, the separation of an FSM kernel may result in increased simulation efficiency.

The kernel is an encapsulation of the *SC_METHOD()* processes along with several member functions to describe an FSM model. In a way it is not necessarily an alternate kernel. However, this encapsulation serves as a step towards isolating the FSM kernel completely from the execution of the DE kernel. At this moment every FSM block executes in one simulation cycle as per our definition of a period for an FSM node. This results in an untimed model of the FSM that will be extended to support timed models in further development. We envision support for timed and untimed models for relevant Models of Computation. Unfortunately, the implementation of signals using *sc_event* types makes it difficult to diverge from the Evaluate-Update semantics. We are currently investigating reconstructing *sc_signal*s such that they can be interpreted by the MoC within which they are employed. This extends the possibility of all MoCs being either timed or untimed. The revamp of the event management is still under investigation.

The Finite State Machine Model of Computation has the following properties:

- A set of states
- A start state.
- An input alphabet and
- A transition function that maps the current state to its next state.

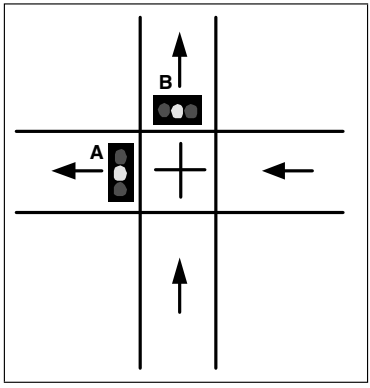


Figure 7.1. FSM Traffic Light Example [4]

FSMs are generally represented in the form of graphs with nodes and transitions connecting the nodes with some conditions on the transitions. Figure 7.1 shows a diagram of a two traffic light system and Figure 7.2 illustrates a Finite State Machine controller for this system.

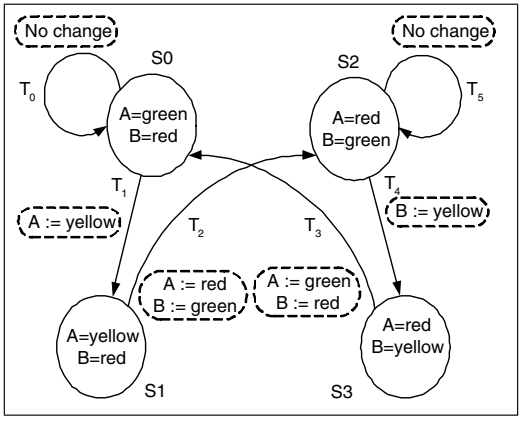


Figure 7.2. FSM Traffic Light Controller Example [4]

The two traffic lights are represented by A and B and the set of states contains S0, S1, S2 and S3. The transitions are represented by the

Table 7.1. Example of *map<...>* data structure

Key	Value
toplevel.state.state0	0xf000001
toplevel.state.state1	0xf000011
toplevel.state.state2	0xf000101
toplevel.state.state3	0xf001001
toplevel.state.state4	0xf100001

arrows and the action associated with the transition is marked in the dotted ellipses. Suppose S0 is the initial state. Then a transition to S1 causes traffic light A to change from green to yellow and B to remain at red. This is a simple controller example, but FSMs can be extensive and large in size. We do not discuss the specifics of Moore and Mealey machines since FSMs serve as pedestals to most engineering. However, we refer the reader to [10] for additional reference and continue our discussion to the implementation details of the FSM kernel in SystemC.

1. Implementation Details

1.1 Data Structure

The FSM kernel's data structure implements a *map<...>* object from the C++ STL. A map object is simply a list of pairs consisting of a key and a value. FSM uses a *string* and a pointer to the *SC_METHOD()* process via the *sc_method_process* class as shown in Listing 7.1 as a pair entry in the *map<...>* object. For illustration purposes Table 7.1 displays the pairs inserted in the data structure. The addresses for the values are made up. The keys are of type *string* and the value is an address to an object of type *sc_method_handle*. The key field is used when searching this *map<...>* object for a particular string and if a pair entry is found with the corresponding search string, then the value is returned.

The *FSMReceiver* class once again derives from the *baseReceiver* class. The *baseReceiver* holds the type of the receiver that is derived from it. Besides the *fsmlist* private data member, there is an *id* and a string type variable called *currentState*. This *currentState* variable preserves the current state that the simulation has reached for the FSM. This may not seem necessary in a pure FSM model. However, in heterogeneous models, a particular state in the FSM may resume another MoC and return back to the FSM requiring the preservation of the last state that it had executed. The member functions of class *FSMReceiver* are standard functions used to insert elements into the *fsmlist* and retrieve a particu-

Listing 7.1. class *FSMReceiver*

```
class FSMReceiver : public baseReceiver {
  private:
    map<string, sc_method_handle> * fsmlist;
    string id;
    string currentState;

  public:
    FSMReceiver(const string & s);
    ~FSMReceiver();

    void insert(const string &s, sc_method_handle h);
    sc_method_handle find(const string &s);
    bool myid(const string &s);

    void setState(const string & s);
    string & getState();

    void fsm_execute();
    sc_method_handle register_fsm_method( const char* name,
                SC_ENTRY_FUNC entry_fn,
                sc_module* module );
};
```

Table 7.2. Some Member functions of class *FSMReceiver*

Member Function	Purpose
insert(...)	Inserts a pair into *fsmlist*
find(...)	Returns a pointer to the FSM process if the string associated with the name of the process is found
setState(...)	Set the *currentState* with the *string* argument that is passed
getState()	Returns the *currentState*
fsm_execute()	Execute the FSM model
register_fsm_method(...)	Called from a C macro that registers a *sc_method_process* object as an FSM process

lar *sc_method_handle* by supplying a *string*. The *register_fsm_method(...)* function is invoked by the C macro defined by *SC_FSM_METHOD(...)*. Listing 7.2 shows the module construction for *SC_FSM_METHOD(...)*. Table 7.2 lists some of the important member functions of class *FSM-Receiver* and their use.

The *fsm_execute()* member function is responsible for initiating the execution of the FSM model. The simulation begins at the initial state, which is set by the modeler. The modeler can do this by using the *setState(...)* member function to designate one of the states as an initial state. To schedule an FSM process to execute, the *setState(...)* member function is employed. A schedule for an FSM process means changing the *currentState* variable to reflect the name of the next process to

Listing 7.2. Macros used to register FSM processes

```
1 //SC_FSM_METHOD...
2 #define SC_FSM_METHOD(func, mod)
3     fsm_declare_method_process( func ## _handle,
4                                 #func,
5                                 SC_CURRENT_USER_MODULE,
6                                 func, mod )
7 //fsm_declare_method_process
8 #define fsm_declare_method_process(handle, name, module_tag,
      func, mod )
9 sc_method_handle handle;
10 {
11     SC_DECL_HELPER_STRUCT( module_tag, func );
12     handle = mod->register_fsm_method( name,
13                 SC_MAKE_FUNC_PTR( module_tag, func ), this );
14                 sc_module::sensitive << handle;
15     sc_module::sensitive_pos << handle;
16     sc_module::sensitive_neg << handle;
17 }
```

execute. This function sets the *currentState* to the argument that is passed into that function and with the next execution of the FSM; the process with that *string* name is executed. Every time *fsm_execute()* runs, the *currentState* of the FSM model is retrieved and a search is done on the data structure. The *sc_method_process* pointer is returned if an entry is found and then executed. The key entries in Table 7.1 are *sc_method_process* object names. The naming convention preserves SystemC naming conventions by adding a dot between module names. This naming convention is discussed in Chapter 5 with an example. For the FSM kernel the *FSMReceiver* is the most integral class. The remainder of the classes implemented to support the FSM kernel are shown in Listing 7.3. The *FSMkernel* class is responsible for allowing multiple FSM models to simulate together.

Channels and ports specific for the FSM MoC are included with the declarations shown in Listing 7.4. Since there is no specific communication functionality for the FSM MoC, the *FSMport* and *FSMchannel* classes inherit from *sc_moc_port* and *sc_moc_channel* respectively. They exhibit the same behavior as their base classes. The source listing for the base classes is shown in Chapter 4.

2. Example of Traffic Light Controller Model using FSM Kernel in SystemC

To further illustrate our FSM kernel we present an FSM traffic light controller example. Figure 7.2 describes the state diagram of this simple example. Listing 7.5 shows the *SC_MODULE(state)* definition along with its respective entry functions. The entry functions are *state0*,

Listing 7.3. FSMkernel and FSMnode class definition

```
class FSMkernel {

  private:
    vector<FSMReceiver*> * fsms;

  public:
    FSMkernel();
    ~FSMkernel();
    void insert(FSMReceiver * f);
    FSMReceiver* find_fsm(const string & id);
    void fsm_crunch();
};

class FSMnode {

  private:
    sc_method_handle handle;
    string name;

  public:
    FSMnode();
    ~FSMnode();
    void set(const string &s, sc_method_handle h);

};
```

Listing 7.4. Ports and Channels for FSM MoC

```
template <class T>
  class FSMport : public sc_moc_port<T> {};
template <class T>
  class FSMchannel : public sc_moc_channel<T> {};
```

state1, *state2* and *state3* representing the states S0, S1, S2, and S3 respectively. Each of these entry functions are bound to an *SC_METHOD()* process via the *SC_FSM_METHOD()* macro. Registration of the entry functions as FSM processes is performed via this macro. The constructor of *SC_MODULE(...)* remains the same as existing SystemC syntax with the use of *SC_CTOR(...)*. Notice the initial state of the FSM is set within the constructor with *fsm_model→setState("toplevel.state.state0")*. We preserve the naming conventions of SystemC to target the FSM process for execution. However, this requires knowledge of the encapsulating process as well since the naming convention of SystemC concatenates the names by taking the module name, adding a dot character at the end, followed by appending the entry function name. The hierarchy of the module is preserved by preceding with the name of the toplevel module name as shown by *toplevel.state.state0*.

Two instances of *light* are present where *A* represents traffic light A and *B* represents traffic light B. The colors are enumerated by *enum*

Listing 7.5. Module Definition of 'state' in Traffic Light Controller Model

```
1 SC_MODULE(state)  {
2
3   light A;
4   light B;
5   int random;
6
7   void state0() {
8     random = rand();
9     cout << "------------------------------------------------------------"
          << endl;
10    cout << "S0 -- Random value = " << random << endl;
11    A = GREEN;
12    B = RED;
13    printLight(A, B);
14    if (random % 2 == 0)
15      fsm_model->setState("toplevel.state.state1");
16  };
17
18  void state1() {
19    random = rand();
20    cout << "------------------------------------------------------------"
          << endl;
21    cout << "S1 -- Random value = " << random << endl;
22    A = YELLOW;
23    B = RED;
24    printLight(A, B);
25    if (random % 2 == 0)
26      fsm_model->setState("toplevel.state.state2");
27  };
28
29  void state2() {
30    random = rand();
31    cout << "------------------------------------------------------------"
          << endl;
32    cout << "S2 -- Random value = " << random << endl;
33    A = RED;
34    B = GREEN;
35    printLight(A, B);
36    if (random % 2 == 0)
37      fsm_model->setState("toplevel.state.state3");
38  };
39
40  void state3() {
41    random = rand();
42    cout << "------------------------------------------------------------"
          << endl;
43    cout << "S3 -- Random value = " << random << endl;
44    A = RED;
45    B = YELLOW;
46    printLight(A, B);
47    if (random % 2 == 0)
48      fsm_model->setState("toplevel.state.state0");
49  };
50
51  SC_CTOR(state)  {
52    fsm_model->setState("toplevel.state.state0");
53    SC_FSM_METHOD(state0, fsm_model);
54    SC_FSM_METHOD(state1, fsm_model);
55    SC_FSM_METHOD(state2, fsm_model);
56    SC_FSM_METHOD(state3, fsm_model);
57  };
58};
```

Listing 7.6. Module Definition of *top* in Traffic Light Controller Model

```
1 SC_MODULE(top) {
2
3   state *s1;
4   void entry() {
5
6     while(true) {
7       fsm_trigger();
8       wait();
9     }
10  };
11
12  SC_CTOR(top) {
13    s1 = new state("state");
14    SC_THREAD(entry) {
15    };
16  };
17 };
18
19
20 int main() {
21   fsm_model = new FSMReceiver("fsm1");
22   fsm_kernel.insert(fsm_model);
23
24   top tp("toplevel");
25   sc_start(-1);
26
27   return 0;
28 }
```

light RED=1, YELLOW= 2, GREEN=3;. The values for the traffic lights are set followed by a execution of a global function *printLight(...)* that displays status of the lights. Full source is not presented, but is available at our website [36]. The next state is set by using the *setState(...)* function call, which describes the transition presented in Figure 7.2. However, since C++ is a sequential programming language, implementing non-determinism for transitions T_0 and T_5 requires the use of randomization. A simple policy where if the randomly generated number is not zero then the transitions T_0 or T_5 are traversed depending on the current state of the FSM is implemented.

The *top* module is a regular *SC_THREAD()* process with an infinite loop and a single suspension statement. This is to allow the FSM to run infinitely, as expected behavior of a traffic light controller. The model progresses after every cycle due to the *wait(...)* statement. Similar to the SDF MoC implementation, a call to *fsm_trigger(...)* is mandatory to indicate the execution of the FSM kernel. Listing 7.6 shows the module definition for *top* along with definition of *sc_main(...)*.

A global object of type *FSMkernel* holds the *fsm_model* that is to execute. The simulation starts using the *sc_start(...)* function call [Listing 7.6, Line 25].

Chapter 8

SYSTEMC KERNEL APPLICATION PROTOCOL INTERFACE (API)

Designed to contain only one simulation kernel, SystemC does not provide an API to tidily add an extension to the simulation kernel. Nor are we aware of any efforts of providing an API for kernel development in SystemC. The existing simulation kernel is specified in the *sc_simcontext* class along with several global function definitions such as *sc_start(...)*. We provide an API that better incorporates multiple kernels and provides a medium through which kernels can gain access to its counterpart kernels. Once again, our approach is in limiting the changes in the current source by overlaying the current implementation with our API with the introduction of a class called *sc_domains*. The problems that we address by adding this encapsulating API class are as follows:

1 For each implemented kernel, one should be able to execute each of them independent of others.

2 Every implemented kernel must be able to access every other kernel for multi-domain heterogeneous simulation.

1. System Level Design Languages and Frameworks

Our API class that we call *sc_domains* is shown in Listing 8.1 and a class diagram is shown in Figure 8.

The *sc_domains* class contains pointers to each implemented kernel. We present two methods by which a kernel can be represented. The first is by dynamically allocating the kernel such as in [Listing 8.1, Line 45] for the *de_kernel* or by having a list of multiple kernels as in [Listing 8.1, Line 22] for *sdf_domain*. The *sdf_domain* could have been easily

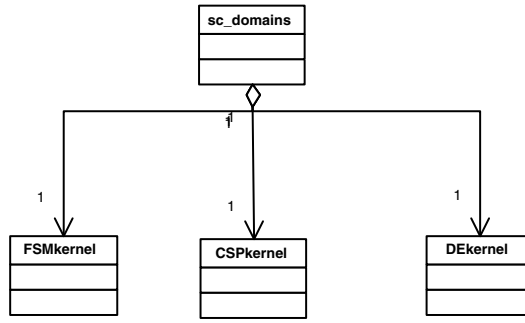


Figure 8.1. Class Diagram for *sc_domains*

implemented as a pointer to a *vector* list, but we choose to do this to show that this approach can also be used. Furthermore, not all kernels require more than one instance valid in the system. For example, the DE kernel requires only one instance of itself. However, there can be multiple SDF or FSM models in a heterogeneous model requiring multiple SDF graphs (SDFG) to be present in the system same applied with multiple FSM models.

Initialization functions and its helper functions must all be member functions of the class *sc_domains*. An example of a helper function is *split_processes()* that is an SDF initialization helper function. Its purpose is to separate SDF method processes from regular DE method processes. Similarly, *find_sdf_graph()* is a helper function that finds a specific SDFG in the entire model.

For a kernel designer it is important to adhere to some general guidelines in adding a kernel to SystemC using the API we provide. Most of these guidelines are intuitive. They are as follows:

1 Ensure that every added kernel is encapsulated in a class such as *sdf_graph*. This class must give enough access to the *sc_domains* class to enable execution of the functions from the *sc_domains* class.

2 Include a pointer (recommended) to an instance of a kernel type as a data member of *sc_domains*. If for some reason there can be multiple instances of the kernel, such as in the case of SDF then a list of pointers to the kernels can be used. Whether to use pointers to kernels or object instances of kernels in implementation is up to the kernel designer.

3 Initialization functions for the kernel must be called from function *init_domains* that is responsible for initializing all kernels. A kernel designer can implement additional functions specific for their kernel

Listing 8.1. API Class *sc_domains*

```
1
2 class sc_domains  {
3
4   // public functions
5   public:
6     sc_domains();
7     ~sc_domains();
8
9     void init_de();
10    void init_sdf();
11    void init_domains(const sc_time & duration, string in);
12
13    void split_processes();
14
15    sdf_graph * find_sdf_graph(string sdf_prefix);
16
17    // take the input from user
18     bool user_input(string in);
19
20    // make these private after the hack works
21    sc_simcontext* sdfde_kernel;  // DE kernel
22    vector<sdf_graph*> sdf_domain; // SDF kernel
23    void sdf_trigger(string topname);
24
25    string user_order;
26
27    // DE functions
28    void initializeDE();
29    bool isDEinitialized();
30
31    // CSP functions
32    void initializeCSP();
33    bool isCSPinitialized();
34
35    // FSM functions
36    void initializeFSM();
37    bool isFSMinitialized();
38
39    void clean_sdf(const string & str);
40    void clear_de();
41    DEkernel * get_de_kernel();
42    CSPkernel * get_csp_kernel();
43
44  private:
45    DEkernel * de_kernel;
46    CSPkernel * csp_kernel;
47    FSMkernel * fsm_kernel;
48 };
```

in *sc_domains*, for example in *sdf_domain*, *split_processes()* is used to split the runnable process list by removing the SDF method processes except for the top-level SDF method process.

Table 8.1 displays some of the member functions in *sc_domains* and their purposes. A file static instance of *sc_domains* is instantiated. The object is called the *Manager*. The API class acts as a manager having access to all models and MoCs that are currently in the system. All kernel instances have access to this *Manager* instance through which they

Listing 8.2. *init_domains* from *sc_domains* class

```
1// initial the domains
2void sc_domains::init_domains(const sc_time & duration, string
     in ) {
3
4  if (in.size() > 0)
5    user_input(in);
6
7  init_de();
8  split_processes();
9  init_sdf();
10 model.sdfde_kernel->de_initialize2();
11}
```

get access to any other kernel implemented and instantiated allowing for interaction between them. This way the *Manager* object can find a particular model in a specific MoC and execute it accordingly. This setup for *sc_domains* exists to allow for behavioral hierarchy for the future.

Table 8.1. Few Member Functions of class *sc_domains*

Functions	Description
sdf_trigger(...)	Global SDF specific function to execute the SDF graph.
init_domains(...)	Function that invokes all initialization member functions for every kernel in *sc_domains*.
split_processes()	SDF specific function to split SDF function block processes from regular SystemC method processes.
init_de()	Create an instance of the DE kernel.
init_sdf()	Initialization function for SDF kernel. Traverses all SDFGs and constructs an *executable schedule* if one exists.
find_sdf_graph(...)	Helper function to find a particular SDF graph for execution.
initializeCSP()	Prepare *csp_kernel* such that instances of CSP models can be inserted
initializeFSM()	Prepare *fsm_kernel* such that instances of FSM models can be inserted
get_de_kernel()	Return the pointer *de_kernel*.
get_csp_kernel()	Return the pointer *csp_kernel*.
get_fsm_kernel()	Return the pointer *fsm_kernel*.

We provide these guidelines to simply help kernel designers in using the introduced API and we impose no restrictions as to a particular method of addition. This *sc_domains* class shown in Listing 8.1 simply encapsulates all the kernel classes requiring a certain alteration to the

Listing 8.3. *init_sdf* from *sc_domains* class

```
1 void sc_domains::init_sdf() {
2
3   if (sdf_domain.size() == 0) {
4     schedule::err_msg(" No SDF system ", "WW");
5     return;
6   }
7
8   for (int sdf_graphs = 0; sdf_graphs < (signed) sdf_domain.
        size() ; sdf_graphs++) {
9
10    // Extract the address of first SDF
11    sdf_graph * process_sdf_graph = sdf_domain[sdf_graphs];
12    // Calculate schedule for this SDFG if one not already
        calculated
13    if (process_sdf_graph->result == NULL)
14      process_sdf_graph->sdf_create_schedule();
15  }
16}
```

Listing 8.4. *sdf_trigger()* from *sc_domains* class

```
1 void sc_domains::sdf_trigger(string topname)  {
2   string sdfname = topname+".";
3   sdf_graph * run_this;
4
5   for (int sdf_graphs = 0; sdf_graphs < (signed) model.
        sdf_domain.size(); sdf_graphs++) {
6     // pointer to a particular SDF graph
7
8     sdf_graph * process_sdf_graph = model.sdf_domain[sdf_graphs
          ];
9
10    if (strcmp(process_sdf_graph->prefix.c_str(), sdfname.c_str
          ())==0 ) {
11
12      run_this = process_sdf_graph;
13
14      if (( run_sdf == true)){
15        // execute the SDF METHODs
16        run_this->sdf_simulate(sdfname);
17        run_sdf = false;
18      }/* END IF */
19    }/* END IF */
20  }/* END FOR */
21}/* END sdf_trigger */
```

global function calls such as *sc_start(...)* and the introduction of MoC specific simulation functions such as *sc_csp_start(...)*. The addition of global and member functions in existing classes are described in the MoC's respective chapter and the API mainly supports the exchange of information about these multi-MoC models.

This API is not the most evolved nor is it the most robust, but it provides a mechanism and an approach to organizing and allowing kernel designers to think and consider additional improvements in managing

their kernel implementation in SystemC. The API is also not fully complete, for example, we do not support multiple models for the CSP MoC as yet. Ideally, we envision the *Manager* object to manipulate the client QuickThread coroutine instances as well, but these considerations are still under investigation.

Finally, Listings 8.2, 8.3 and 8.4 show some of the API functions.

Chapter 9

HETEROGENEOUS EXAMPLES

1. Model using SDF kernel

We construct a heterogeneous example for an image converter shown in Figure 9.1. This system begins by downloading encrypted images that are decrypted, converted to a specific type, then encrypted again and uploaded back to the source. This is a multi-MoC model where the first DE block (Decryption block) is responsible for downloading encrypted images and decrypting them. These images are passed onto the SDF block that performs the Sobel edge detection algorithm on the image. This is our preferred conversion type. Output from Sobel is sent to the final DE block (Encryption block) that encrypts the converted image and uploads it to a particular location. All DE blocks consist of *SC_CTHREAD()* processes and the SDF block uses *SC_METHOD()* processes.

We create this model using three scenarios: a pure DE implementation, a DE implementation such that the processes are non-threaded using the transformation technique in [62] and a heterogeneous implementation using the DE and SDF kernels. The Discrete-Event model of the Converter uses control signals to indicate which process executes next and so on. The non-threaded model converted every *SC_THREAD()* or *SC_CTHREAD()* process to an *SC_METHOD()* using the transformation technique in [62]. The DE-SDF model is shown in Figure 9.1.

An interesting aspect about this model is the interaction between the DE and SDF blocks. By interaction we mean the data transfer. SystemC channels may be employed as the interaction medium between the DE and SDF blocks, but we advise using *SDFport*s and *SDFchannel*s to push the data onto the SDF toplevel. Both the blocks responsible for

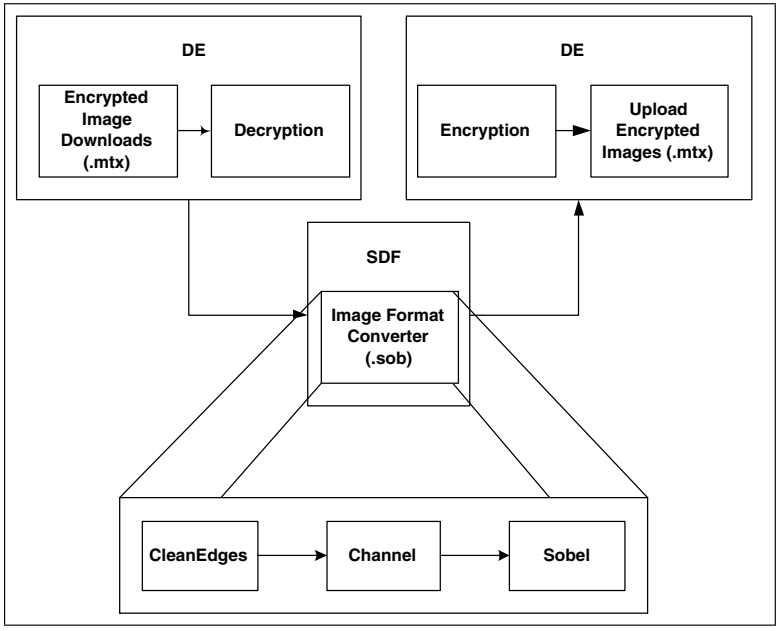


Figure 9.1. Image Converter Diagram

pushing data into the SDF and the SDF itself, must have access to the *SDFchannel* and since MoC-specific channels and ports do not generate SystemC events, this can be done. The block pushing data into the SDF model can also have an *SDFport* that binds to an *SDFchannel* that the SDF component retrieves its inputs from. A control signal can be used to trigger the SDF toplevel process once data is ready on the channel for the SDF to consume. This is a simple example showing how we implement a heterogeneous model (the image Converter) using our improved modeling and simulation framework.

Figure 9.2 shows approximately 13% improvement over the original model. We attribute the limitation in simulation efficiency increase to Amdahl's law [41]. The SDF block in this Converter model serves only a small portion of the entire system allowing for only that much improvement in total simulation performance. If the SDF component was responsible for more percentage of the original model, then the simulation efficiency of that model would be significantly larger than its counterpart (created using the DE kernel).

We profiled the Converter model. Taking an approximate percentage of time spent for the SDF model when using the original reference implementation kernel, we can see from Table 9.1 that the approximated total running time for the SDF is approximately 14%. This particular

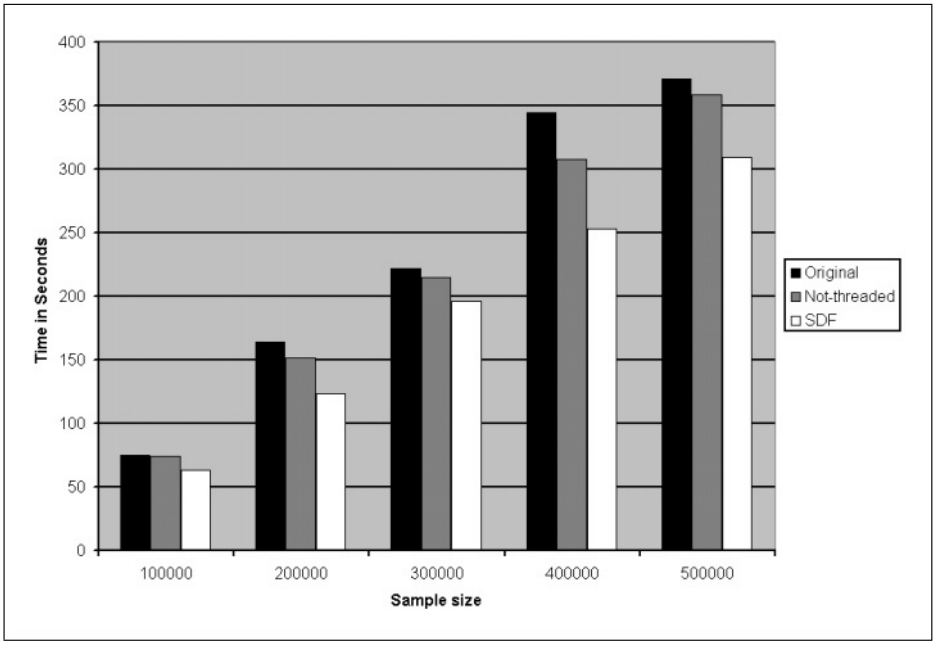


Figure 9.2. Converter Results

implementation of the Converter model has *sc_fifo* channels that generate numerous events during passing of data. Now, using Amdahl's law from Equation 9.1 and using 14% as the fraction of enhanced model we can follow Equation 9.1 [41] where s is the speedup of the enhanced portion. This equation implies that, if we were to enhance the simulation of the SDF components by 2x, then we can only achieve a simulation performance less than $\frac{0.14}{2} = 7\%$. Therefore, if our SDF components was either larger or if we have more SDF components in a model then the simulation performance will improve according to the percentage of SDF component in the model.

$$Overall\ Speedup = \frac{1}{((1 - .14) + .14/s)} \tag{9.1}$$

Table 9.1. Profiling Results for Converter Model

Function	% of total running time spent
cleanedges::cleanedge(void)	6.78
sobel::operate(void)	5.66
channel::latch(void)	1.18

2. Model using CSP and FSM kernels

We reuse the Dining Philosopher example introduced in Chapter 6 to construct a heterogeneous model using the CSP and FSM kernels. The model previously created was a pure CSP model, where every philosopher and fork was a CSP process. We mentioned that a superficial implementation of the Dining Philosopher problem can result in deadlock, to which we presented the idea of a footman. The role of the footman is to seat the philosophers to their designated seats and to monitor that only four philosophers are sitting at the dining table at any time. The implementation of the footman was done via a global function to have immediate verification of the state of the seats occupied at the dining table. However, we take this example further by implementing the footman as a Finite State Machine controller embedded in a CSP process.

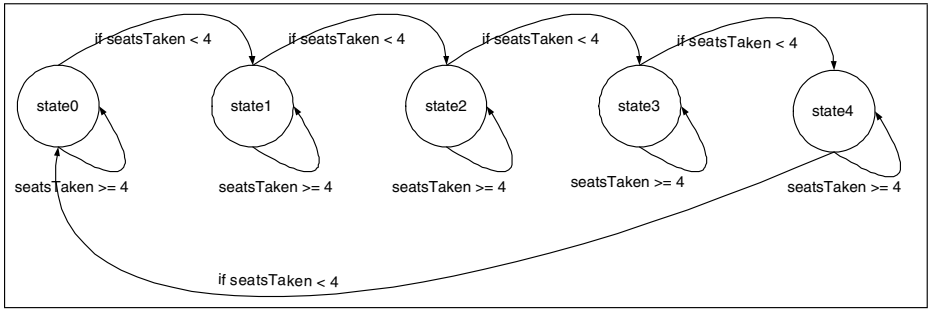


Figure 9.3. FSM implementation of the Footman

The FSM based footman allows the seating of four philosophers, where by each seat allocation signifies a state. A global seat counter called *seatsTaken* is updated every time a philosopher is assigned a seat and when a philosopher leaves his seat. The FSM in Figure 9.3 is combined with the Dining Philosopher implementation to yield a heterogeneous example shown in Figure 9.4.

Figure 9.3 presents a state machine diagram showing the functions of the footman. The initial state is state0. The functionality of every state is the same except for the next state transitions. Every state has a self-loop suggesting that the control in the FSM does not transition to another state whenever four philosophers are seated. However, if there are seats available, then a seat is allocated and the transition to the next state occurs. This is a simple FSM that changes the solution of the Dining Philosopher such that it ensures that every philosopher gets a turn to eat as well as serving as a deadlock avoidance mechanism. The module definition is shown in Listing 9.1 where object *s* is the

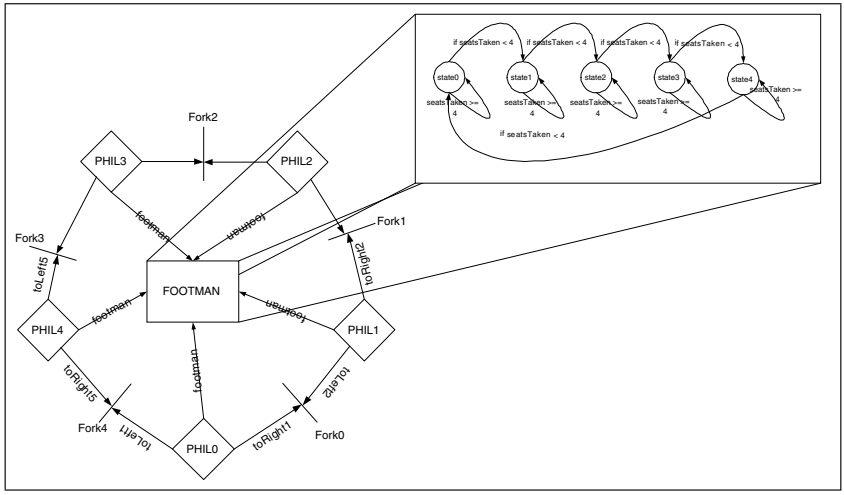


Figure 9.4. Dining Philosopher Model with FSM footman

Listing 9.1. Module definition Footman FSM

```
1 SC_MODULE(s) {
2
3   int random;
4   int * giveSeat;
5
6   CSPnode * csp;
7   CSPport<int> * fromPhil0;
8   CSPport<int> * fromPhil1;
9   CSPport<int> * fromPhil2;
10  CSPport<int> * fromPhil3;
11  CSPport<int> * fromPhil4;
12
13  void state0();
14  void state1();
15  void state2();
16  void state3();
17  void state4();
18
19  SC_CTOR(s) {
20    giveSeat = new int();
21    *giveSeat =1;
22    fsm_model->setState("toplevel.state.state0");
23    SC_FSM_METHOD(state0, fsm_model);
24    SC_FSM_METHOD(state1, fsm_model);
25    SC_FSM_METHOD(state2, fsm_model);
26    SC_FSM_METHOD(state3, fsm_model);
27    SC_FSM_METHOD(state4, fsm_model);
28  };
29 };
```

state machine. This module itself has pointers to *CSPnode* and *CSPport* types. The variables with prefix *fromPhil* are the ports through which the philosophers communicate with the footman requiring the footman

to be encapsulated in a CSP process. Therefore, the FSM defining the behavior of the footman is embedded in a CSP process through which the *CSPnode* object and *CSPchannel*s are tunneled. The rendez-vous communication occurs from the toplevel process encapsulating the FSM controller. Pointers to objects of type *CSPchannel* and *CSPnode* are necessary for the FSM to have access to the *CSPchannel*s that it must communicate with and make function calls on the *CSPnode* object.

The implementation of the state entry functions are shown in Listing 9.2. Every state has identical implementation except for the next state transition.

Listing 9.2. State entry functions for Footman FSM

```
1 void s::state0() {
2   if (seatsTaken < 4) {
3     ++seatsTaken;
4     seatAvailable[0] = true;
5     fromPhil0->push(*giveSeat, *csp);
6     fsm_model->setState("toplevel.state.state1");
7   }
8 };
9
10 void s::state1() {
11   if (seatsTaken < 4) {
12     ++seatsTaken;
13     seatAvailable[1] = true;
14     fromPhil1->push(*giveSeat, *csp);
15     fsm_model->setState("toplevel.state.state2");
16   }
17 };
18
19 void s::state2() {
20   if (seatsTaken < 4) {
21     ++seatsTaken;
22     seatAvailable[2] = true;
23     fromPhil2->push(*giveSeat, *csp);
24     fsm_model->setState("toplevel.state.state3");
25   }
26 };
27
28 void s::state3() {
29   if (seatsTaken < 4) {
30     ++seatsTaken;
31     seatAvailable[3] = true;
32     fromPhil3->push(*giveSeat, *csp);
33     fsm_model->setState("toplevel.state.state4");
34   }
35 };
36
37 void s::state4() {
38   if (seatsTaken < numPhil - 1) {
39     ++seatsTaken;
40     seatAvailable[4] = true;
41     fromPhil4->push(*giveSeat, *csp);
42     fsm_model->setState("toplevel.state.state0");
43     }
44   };
```

The *seatAvailable* array maintains which seat has been occupied and a record of every philosopher to his particular seat is kept by the index of the array. For example, *seatAvailable[1]* refers to the seat that belongs to a philosopher with *id* one.

Listing 9.3. Toplevel CSP process for Footman

```
1 SC_MODULE(fsmtop) {
2
3   s * s1;
4
5   CSPnode csp;
6   CSPport<int> fromPhil0;
7   CSPport<int> fromPhil1;
8   CSPport<int> fromPhil2;
9   CSPport<int> fromPhil3;
10  CSPport<int> fromPhil4;
11
12  void entry();
13
14  SC_CTOR(fsmtop) {
15    s1 = new s("state");
16    s1->csp = &csp;
17    s1->fromPhil0 = &fromPhil0;
18    s1->fromPhil1 = &fromPhil1;
19    s1->fromPhil2 = &fromPhil2;
20    s1->fromPhil3 = &fromPhil3;
21    s1->fromPhil4 = &fromPhil4;
22
23    SC_CSP_THREAD(entry, DP, csp);
24  };
25 };
```

The toplevel CSP process contained in *fsmtop* module is defined in Listing 9.3. This module definition has instances of *CSPnode* and *CSPport*s that had pointer declarations in Listing 9.1. The constructor of module *fsmtop* initialize an instance of *s* and appropriately assigns the addresses of the ports in object *s1*.

The model with the addition of the footman CSP process is shown in Figure 9.4. This pictorial representation of the Dining Philosopher also shows *CSPchannel*s from every philosopher to the footman. This addition requires alteration to the main entry function for the philosopher. This change requests a seat from the footman before proceeding with the *getfork()* function to request forks. The only addition is calling *footman.get(...)* such that the philosopher suspends itself until the footman process is executed and a seat is allocated. The FSM controller manages the seat allocation and returns a value on the channel designating a seat, resuming the suspended philosopher process.

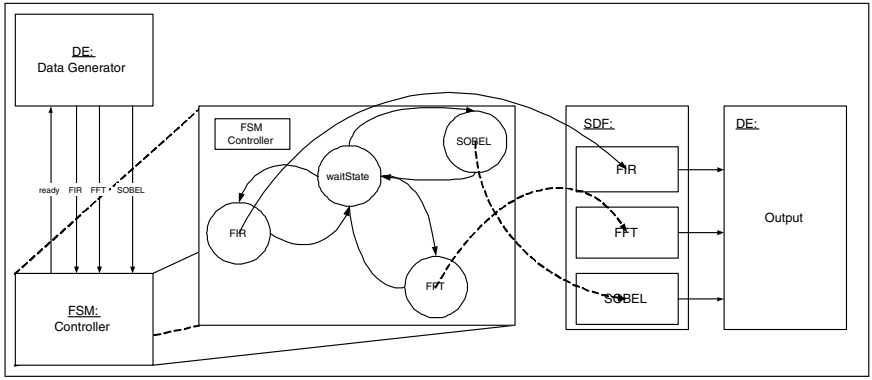


Figure 9.5. Heterogeneous Example using FSM, SDF and DE Models

3. Model using FSM, SDF and DE kernels

Another heterogeneous example is shown in Figure 9.5. This model is separated into DE, FSM, SDF and DE blocks as shown in the diagram. The DE models are the data generator and the output, the SDF models are the FIR, FFT and Sobel, and the FSM model is the controller responsible for triggering the SDF computations. The first DE block is the data generator, which at random selects one of the three SDF models for execution. It also uses the SystemC Verification library (SCV) [49] for generating randomized input data for the FIR and FFT. Input for the Sobel is read from a file. The data generator block sends a signal to the FSM controller to execute the chosen SDF computation.

The FSM model consists of four states. The initial state is the *waitState* and the other three states fire the FIR, FFT, and Sobel SDF models respectively. The *waitState* receives a signal from the data generator block and according to the signal, transition to the respective state is taken. For example, suppose the data generator block sends a signal to execute the FIR model, then the FSM takes a transition from the *waitState* to the *FIR* state. The SDF model of the FIR executes when in this state. Upon complete execution of the SDF FIR model, control returns back to the *FIR* state that takes a transition back to the *waitState* state. In turn, the FSM controller sends a signal back to the data generator that the controller is ready to receive another input.

This example illustrates the use of SCV with a heterogeneous MoC employing MoC-specific kernels. Invocations of an FSM model, DE model and SDF models are also presented in this example. However, due to the lack of space the source code is made available at [36].

4. Model using CSP, FSM, SDF and DE kernels

Building on top of the example with an FSM based footman for the Dining Philosophers problem, we construct another model that uses the SDF and DE kernels along with the CSP and FSM. This example shows the use of all MoCs implemented in SystemC until now. In the model presented in Figure 9.6, we use the Sobel edge detection model as our SDF component, the RSA encryption algorithm and the Producer/Consumer FIFO example as the DE components (from SystemC distribution examples) and the footman as the FSM controller.

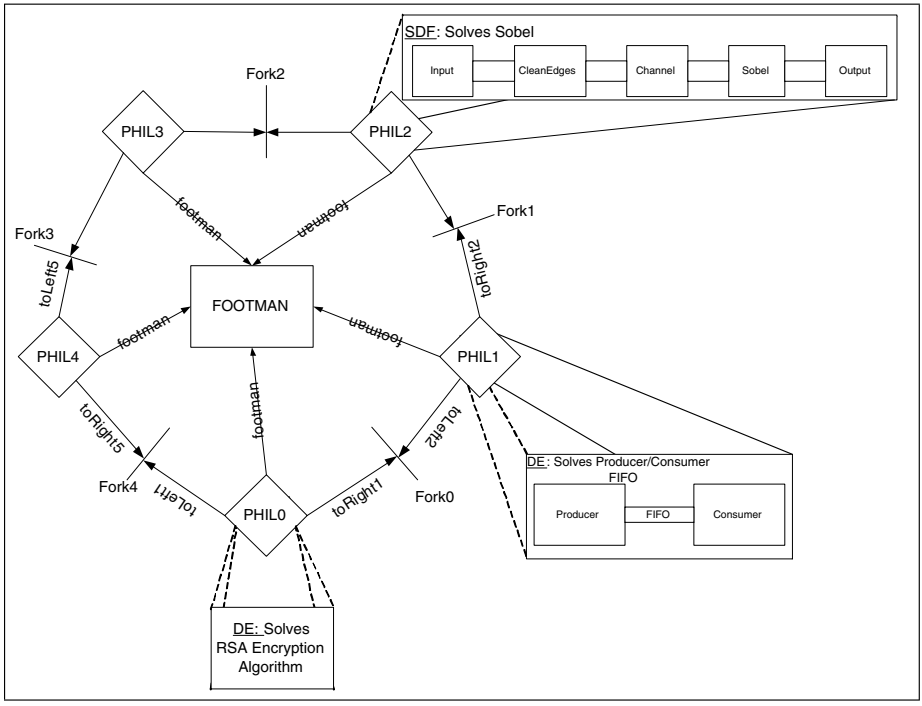


Figure 9.6. Truly Heterogeneous Dining Philosopher Model

The footman FSM controller is encapsulated in a CSP process but the DE and SDF models are not. The roles of the philosophers when in the thinking state have been slightly altered with this implementation and they are as follows: Phil0 solves an RSA encryption algorithm, Phil1 simulates the Producer/Consumer example using the FIFO implementation, Phil2 performs the Sobel edge detection, and Phil3 and Phil4 sit around just pretending that they are thinking. Listing 9.4 shows the *PHIL* module entry function with the added invocations to the DE and SDF kernels.

Listing 9.4. Entry function for PHIL module

```
void PHIL::soln() {
 int duration = _timeToLive;
 int eatCount = 0;
 int totalHungryTime = 0;
 int becameHungryTime;
 int startTime = msecond();
 while ( 1 ) {
   seatAvailable[id] = false;
   int got = footman.push(csp);
   if ((seatAvailable[id] == true) && ((state[id] != 0) || (
       state[id] != 6)) ) {
     becameHungryTime = msecond();
     print_states();
     getfork();
     print_states();
     totalHungryTime += ( msecond() - becameHungryTime );
     eatCount++;
     state[id] = 3;
     usleep( 1000L * random_int( MeanEatTime ) );
     print_states();
     dropfork();
     usleep( 1000L * random_int( MeanThinkTime ) );
     print_states();
     state[id] = 6;
     if (id == 1 ) {
       rsa(-1);
       sc_de_init("rsa");
       sc_de_trigger(-1, "rsa");
     }
     if (id == 2 ) {
       top top1("Top1");
       sc_de_init("Top1");
       sc_de_trigger(-1, "Top1");
     }
     if (id == 3) {
       toplevel sdftop("sdftop");
       sc_de_init("sdftop");
       sc_sdf_start(1,"sdftop");
     };
     usleep( 1000L * random_int( MeanThinkTime ) );
     state[id] = 0;
     print_states();
     --space;
     --seatsTaken;
     csp.reschedule();

   } else {
       csp.reschedule();
   }
 }
 state[id] = 7;
 };
```

We compare *id* to allow for a specific philosopher to perform certain tasks during their thinking phase. We mandate that every model must be encapsulated with a toplevel process. We use *SC_METHOD()* processes since we want single execution of the model. Our SDF kernel is an untimed MoC implementation, however DE examples can span for a certain length of time. To support this we added global functions

sc_de_trigger(...) and *sc_sdf_trigger(...)*. Both these functions take in the duration of the execution and the name of the toplevel process that is to be executed. For the SDF, the duration has no effect. For the DE, the simulation runs for the specified duration after which control returns to the calling thread. The *sc_de_init(...)* initializes the model specified by the argument passed into the function to be executed. This same initializer function is used to insert the SDF model.

Chapter 10

EPILOGUE

In this epilogue, we reemphasize the importance of creating multi-MoC extensions of SystemC one more time, and summarize some of the contributions and goals of this book. The basic objective of this endeavor has been to disseminate our experience in extending SystemC with a multi-MoC kernel implementation, and the implementation itself.

One of the major advantages of SystemC over other forms of hardware description languages is the full expressive power of C++ at the designers disposal while modeling in SystemC. However, this advantage often shows the flip side of the coin, becoming a disadvantage. The freedom of using any C++ construct introduces programming errors, and often leads designers to use C++ constructs that are not synthesizable as hardware. Moreover, the lack of structure for creating models for specific MoCs leads to lack of *fidelity*. The term *fidelity* of a modeling framework in this case refers to an informal measure of how accurately one can model behaviors specific to an MoC using the modeling structures and facilities available in the framework. Besides fidelity, not exploiting the MoC specific properties of models implies less efficient simulation, as we have shown in the case of SDF models.

When SystemC was introduced in September 1999, there were two main selling points discussed in the industry and academia. First, SystemC is a class library based on C++, and hence any standard C++ compiler can create executables from SystemC models, which implies free simulation platform rather than expensive VHDL or Verilog simulator. Second, the flexibility of C++ allows designers to be creative, and using C++ makes it easier for software/hardware co-simulation, avoiding PLIs which incur lots of overhead during simulation.

It was quickly realized by industry practitioners that (i) using C++ by itself is not going to provide faster simulation, and (ii) free use of C++ is more of a liability than advantage, as synthesizability becomes an issue with arbitrary C++ constructs. In fact, if the level of abstraction remains at the RTL level, using C++, or using the industry best HDL simulators provide almost equivalent performance. However, if one has to synthesize hardware from SystemC model, it is almost necessary to remain at the RTL level of abstraction barring a few exceptions.

In 2001, SystemC-2.0 was introduced with some radical new features, and it borrowed a lot of concepts from the SpecC language. The most notable of those were the idea of channels, events, and interfaces. The idea of communication refinement, transaction level modeling, and interface based design were motivating factors for such changes. However, transaction level models are difficult to synthesize from, and tools that are commercialized since then can synthesize efficient hardware only from very limited set of constructs. Most problematic with such evolution of SystemC has been that heterogeneous and multi-MoC modeling does not have a direct support in SystemC-2.x yet.

Although we have discussed this extensively throughout this book, we would discuss this again here. Current SystemC simulation kernel is geared towards Discrete-Event (DE) simulation semantics, incorporating delta cycles which is very appropriate to model RTL level digital hardware, but not suitable for other Models of Computation. One can model any other Model of Computation by programmatic innovations, but eventually the simulation targeted kernel has the DE simulation semantics. This kernel uses dynamic event scheduling, and delta events to trigger delta cycles. For models which naturally belong to other alternative MoCs and are amenable to static scheduling, or other kinds of optimizations, when mapped to a DE kernel become inefficient in their simulation timing. So we believe that the only way SystemC can be made a full fledged system level design language is to enable SystemC to handle multi-MoC modeling and simulation, and support for behavioral hierarchy. This will allow designers to model systems which consist of heterogeneous components, modeled in different MoCs, and are hierarchically described. Moreover, the simulation of such models should not require flattening of hierarchy.

The reason why such heterogeneous modeling and simulation is important becomes clear if one looks at any embedded system or a System-on-Chip that goes in a consumer electronics equipment today. For example, a digital camera chip would consist of DSP, microcontrollers, A/D and D/A converters, memory elements and so on. Such systems are conglomerates of components best modeled in multiple MoC domains.

Embedded software or real-time light-weight operating system stacks could also be modeled in a heterogeneous modeling framework, which is currently difficult with SystemC.

SystemC standards body, and open SystemC initiative (OSCI) have been working over the last few years to make changes to SystemC standards to accommodate some of these needs. SystemC-AMS or SystemC-4.0 is slated to incorporate the libraries that will allow continuous domain modeling, which will facilitate the modeling of Analog and Mixed Signal Components. We are also aware of some activities related to software APIs for modeling embedded software in SystemC-3.0, but we are not aware of the current status of these efforts.

However, adding more libraries is not necessarily the solution to the problem at hand. Our belief is that once we create ways to adjoin multiple MoC specific simulation kernels and modeling constructs to SystemC, we will not only enable heterogeneous modeling, but also enable designers of SystemC based tools to easily add capabilities that are planned for SystemC-3.0 or SystemC-4.0.

We have therefore gone ahead, and created our prototype for extensions of SystemC-2.0.1 that enables us to create multi-MoC models, and allows us to exploit through the features of these enhanced kernels the MoC specific properties of these models to obtain simulation efficiency. In particular, in this book we have presented three MoC specific kernels, SDF, CSP and FSM, which we thought were very important MoCs for many embedded system components. Since our effort is limited by personnel and funding, we have not been able to provide a full industrial strength system, but we are putting forth a prototype-scale proof of concept. We have implemented three kernels, created some APIs that will allow others to add their own MoC specific kernels and functionalities, and we have created some heterogeneous models that use these kernels in conjunction. We have also shown efficiency gain in case of SDF, but due to lack of resources we have not done benchmarking for the CSP or FSM kernels, but we believe that with proper experimentation it would be easily revealed that exploiting MoC specific properties can only enhance the simulation performance.

Our hope is that this book would be able to convince some industry groups that multi-MoC extensions of SystemC is not only justified, it is necessary for SystemC to become a true system level modeling language. If our prototype can spark discussions within the SystemC community regarding the usefulness of such extensions, and about the best ways to implement such extensions, we would feel that our endeavor has been amply rewarded. Our implementation specifics of the design of the kernels may not be the only way or the best possible way to achieve these

extensions, but it is one of the many possibilities. We urge the readers of the book to download our prototype, experiment with it, and send us comments and feedback [36].

References

[1] D. Abraham, H. D. Patel, and S. K. Shukla, *A Multi-MOC Framework for SOC Modeling using SML*, FERMAT Lab Tech Report 2004-04, 2004.

[2] Eklectic Ally, *Simulation Engine Example*,
Website: http://eklectically.com/home.html.

[3] J. Armstrong, *Ptolemy Eye Model*,
Website: http://www.ee.vt.edu/∼pushkin/ece6444/presentation_armstrong.pdf.

[4] D. Berner, S. Suhaib, S. Shukla, and H. Foster, *XFM: Extreme Formal Method for Capturing Formal Specification into Abstract Models*, Tech. Report 2003-08, Virginia Tech, 2003.

[5] S. Bhattacharyya, P. Murthy, and E. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, 1996.

[6] Shuvra S. Bhattacharyya, Elaine Cheong, John Davis II, Mudit Goel, Christopher Hylands, Bart Kienhuis, Edward A. Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, John Reekie, Neil Smyth, Jeff Tsay, Brian Vogel, Winthrop Williams, Yuhong Xiong, Yang Zhao, and Haiyang Zheng, *Heterogenous Concurrent Modelling and Design in Java: Volume 2 - Ptolemy II Software Architecture*, Memorandum UCB/ERL M03/28, July 2003.

[7] S. Borkar, *Design challenges of technology scaling*, In IEEE Micro **19** (1999), 23–29.

[8] N. Chomsky, *Three models for the description of language*, IRE Transaction on Information Theory **2** (1956), no. 3, 113–124.

[9] A. Church, *The calculi of lamdba conversion*, Princeton University Press, 1985.

[10] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*, The MIT Press, 1999.

[11] M. Clausen and A. Fortenbacher, *Efficient solution of linear Diophantine equations*, Journal of Symbolic Computation **8** (1989), no. 1-2, 201–216.

[12] National Research Council, *Embedded Everywhere*, National Academy Press, 2001.

[13] CPPreference, *Cppreference*, Website: htpp://www.cppreference.com.

[14] F. Doucet, R. Gupta, M. Otsuka, P. Schaumont, and S. Shukla, *Interoperability as a design issue in c++ based modeling environments*, Proceedings of the 14th international symposium on Systems synthesis, 2001.

[15] A. Jantsch et al., *The ForSyDe Project*,
Website: http://www.imit.kth.se/forskningsprojekt-detalj.html?projektid=46.

[16] International Technology Roadmap for Semiconductors,
Website: http://www.itrs.net/.

[17] FORTE, *Forte Design Systems*, Website: http://www.forteds.com/.

[18] A. Fortenbacher, *Algebraische unifikation*, Master's thesis, Universitat Karlsruhe, 1983.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, October 1994.

[20] P. Garg, S. K. Shukla, and R. K. Gupta, *Efficient Usage of Concurrency Models in an Object-Oriented Co-design Framework*, In Proceedings of DATE '01, 2001.

[21] GNU, *Autoconf*, Website: http://www.gnu.org/software/autoconf/.

[22] ———, *Automake*, Website: http://www.gnu.org/software/automake/.

[23] T. Grotker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*, Kluwer Academic Publishers, 2002.

[24] Ptolemy Group, *HTVQ Block Diagram*,
Website: http://ptolemy.eecs.berkeley.edu/ptolemyII/ptII1.0/.

[25] ———, *Ptolemy II*,
Website: http://ptolemy.eecs.berkeley.edu/ptolemyII/.

[26] T. Guckenbiehl and A. Herold, *Solving Linear Diophantine Equations*, Tech. Report SEKI-85-IV-KL, Universitat Kaiserslautern, 1985.

[27] C. Hoare, *Communicating Sequential Processes*, Communications of the ACM, 21, vol. 8, ACM Press, 1978, pp. 666–677.

[28] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.

[29] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[30] G. Huet, *An algorithm to generate the basis of solutions to homogeneous linear Diophantine equations*, Information Processing Letters, April 1978, 7(3).

[31] J. Jájá, *Introduction to Parallel Algorithms*, Addison-Wesley, 1992.

[32] A. Jantsch, *Modeling Embedded Systems And SOC's - Concurrency and Time in Models of Computations*, Morgan Kaufmann Publishers, 2003.

[33] J.Walker, *The Analytical Engine: The First Computer*, Website: http://www.fourmilab.ch/babbage/, 2004.

[34] G. Kahn, *Coroutines and networks of parallel processes*, Information Processing, North-Holland Publishing Company, 1977.

[35] David Keppel, *Tools and Techniques for Building Fast Portable Threads Packages*, Tech. Report UWCSE 93-05-06, University of Washington Department of Computer Science and Engineering, May 1993.

[36] FERMAT Research Lab., *SystemC-H Website*, Website: http://fermat.ece.vt.edu/systemc-h/.

[37] L. Lavagno, A. Sangiovanni-Vincentelli, and E. Sentovich, *Models of Computation for Embedded System Design*, Website: citeseer.nj.nec.com/lavagno98model.html, 1998.

[38] E. A. Lee and D. G. Messerschmitt, *Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing*, In Proceedings of IEEE Transactions on Computers, NO. 1, vol. Vol. C-36, 1987.

[39] Edward A. Lee and Alberto L. Sangiovanni-Vincentelli, *Comparing Models of Computation*, In Proceedings of the International Conference on Computer-Aided Design (ICCAD), IEEE Computer Society, 1996, pp. 234–241.

[40] Formal Systems (Europe) Ltd., *The FDR Model Checker*, Website:http://www.fsel.com/, 2004.

[41] A. Michalove, *Amdahl's Law*, Website: http://home.wlu.edu/∼whaleyt/classes/parallel/topics/amdahl.html.

[42] MicroLib, *PowerPC 750 Simulator*, Website: http://www.microlib.org/G3/PowerPC750.php.

[43] R. Milner, *Communication and concurrency*, Prentice Hall, 1989.

[44] Gordon Moore, *Cramming more Components onto Integrated Circuits*, **38** (1965), no. 8.

[45] B. Niemann, F. Mayer, F. Javier, R. Rubio, and M. Speitel, *Refining a High Level SystemC Model*, Kluwer Academic Publishers, 2003, In SystemC: Methodologies and Applications, Ed. W. Muller and W. Rosenstiel and J. Ruf.

[46] NS-2, *Network Simulator 2*, Website: http://www.isi.edu/nsnam/ns/.

[47] Sobel operator, *Sobel Operator Algorithm*, Website: http://www.visc.vt.edu/armstrong/ee5514/as4_00.pdf.

[48] OPNET, *OPNET*, Website: http://www.opnet.com/.

[49] OSCI, *SystemC* , Website: http://www.systemc.org.

[50] C. H. Papadimitriou, *Computational complexity*, Addison-Wesley, 1994.

[51] D. Pasechnik, *Linear Diophantine Equation Solver*, Website: http://www.thi.informatik.uni-frankfurt.de/∼dima/software.html.

[52] H. D. Patel, *HEMLOCK: HEterogeneous ModeL Of Computation Kernel for SystemC*, Master's thesis, Virginia Polytechnic Institute and State University, December 2003, Website: http://fermat.ece.vt.edu/hiren/thesis.pdf.

[53] H. D. Patel and S. K. Shukla, *Towards A Heterogeneous Simulation Kernel for System Level Models: A SystemC Kernel for Synchronous Data Flow Models*, Tech. Report 2003-01, FERMAT Lab Virginia Tech., 2003.

[54] _____, *Towards A Heterogeneous Simulation Kernel for System Level Models: A SystemC Kernel for Synchronous Data Flow Models*, Proceedings of International Symposium in VLSI, IEEE Computer Society Press, 2004.

[55] _____, *Towards A Heterogeneous Simulation Kernel for System Level Models: A SystemC Kernel for Synchronous Data Flow Models*, Proceedings of Great Lakes Symposium in VLSI, 2004.

[56] _____, *Truly Heterogeneous Modeling with SystemC*, ch. Formal Models and Methods for System Design, Kluwer Academic Publishers, The Netherlands, 2004.

[57] B. Pierce, *Types and Programming Languages*, MIT Press, 2002.

[58] M. O. Rabin and D. Scott, *Finite Automata and Their Decision Problems*, IBM Journal of Research **3** (1959), no. 2, 115–125.

[59] D. Sangiorgi and D. Walker, *The pi-calculus: A theory of mobile processes*, Cambridge University Press, 2003.

[60] J. R. Senning, *Solution of the Dining Philosophers Problem using Shared Memory and Semaphores*, Website: http://www.math-cs.gordon.edu/courses/cs322/projects/p2/dp/, 2000.

[61] S. Sharad, D. Bhaduri, M. Chandra, H. Patel, and S. Syed, *Systematic Abstraction of Microprocessor RTL models to enhance Simulation Efficiency*, In Proceedings of Microprocessor Test and Verification 2003, 2003.

[62] S. Sharad and S. K. Shukla, *Efficient Simulation of System Level Models via Bisimulation Preserving Transformations*, Tech. report, FERMAT Lab Virginia Tech., 2003-07.

[63] SPECC, *SpecC*, Website: http://www.ics.uci.edu/specc/.

[64] Synopsys, *Synopsys*, Website: http://www.synopsys.com/.

[65] SystemVerilog, *System Verilog*, Website: http://www.systemverilog.org/.

[66] A. M. Turing, *On computable numbers with an application to the entscheidungs problem*, Proceedings of London Mathematical Society **2** (1936), no. 42, 230–265.

[67] V. A. Uspensky, *Post's machine*, Firebird Publications, Inc., 1983.

[68] VERILOG, *Verilog*, Website: http://www.verilog.com/.

[69] VHDL, *VHDL*, Website: http://www.vhdl.org/.

[70] M. H. Weik, *The ENIAC Story*,
Website: http://ftp.arl.mil/∼mike/comphist/eniac-story.html, 1961.

Appendix A
QuickThreads in SystemC

Extension of SystemC for hierarchy and heterogeneity necessitates the understanding of QuickThread packaging and implementation before alteration. Unfortunately, the existing SystemC source does not provide much internal documentation to assist kernel experimenters with a better understanding of SystemC's QuickThread implementation. In this section, we explain the QuickThread implementation such that motivated kernel developers for SystemC can quickly comprehend the specifics. We discuss QuickThreads in general and urge readers to refer to [35]. We continue by describing how threads in SystemC employ the QuickThread client package and how it enforces certain syntax requirements. We expect the reader to be familiar with basic SystemC constructs.

1. QuickThreads

QuickThread (QT) is a package core and not a standalone thread package by itself. The difference is that a package core only provides an interface to create machine dependent code for easier portability, whereas a standalone thread package provides the user with full implementation of stack space, thread synchronization and sometimes even scheduling. Instead, QuickThread allows users to construct non-preemptive thread packages. The goals of the QuickThread core package are as follows:

- To provide an easy API to construct user-level thread packages. These user thread packages can be seamlessly ported to architectures supported by QuickThread.
- To separate execution of threads from their allocation and scheduling.

Synchronization of threads for uni and multi processors has problems of its own such as race conditions, violation of stack space access, need for locking, mutexes, extra context switches and so on. We do not discuss these issues here since our focus is in gaining some basic level understanding of QuickThreads and their implementation with SystemC. The QT package uses *preswitch* as its synchronization mechanism. This mechanism functions in the following manner:

1 Block the current executing thread.

2 Switch to the new thread's stack and execute some clean-up code for the old thread.

The immediate disadvantage of this approach is that a thread cannot context switch to itself, because the *preswitch* synchronization requires a new thread to execute some clean-up code on the old thread. Hence, a new thread must execute for the old thread to be completely switched out. Furthermore, QuickThread does not implement any locking, making the users implement a locking mechanism. The user packaging of QuickThread will be referred to as the client package [35].

A QuickThread can have several states that distinguish the modes of the thread during its lifetime and they are as follows:

- Uninitialized: A thread that requires stack allocation to be performed.
- Initialized: Stack allocation is performed on the thread and a function and its arguments are initialized on the stack region.
- Ready-to-run: Once initialized, the thread is ready-to-run. This is the same as a thread that is Blocked.
- Running: The thread scheduled on the processor for execution.
- Blocked: The suspended thread.

A thread is created when the client code allocates stack space for the particular thread. The stack can either grow upwards or downwards depending on the machine architecture, which has to be handled by the client package. The client thread allocation routine passes in the address and size of the stack space to a QuickThread routine that returns a stack pointer of the uninitialized thread. This is when the thread is in its uninitialized state. For the client to initialize this thread, a QuickThread initialization primitive is used to initialize the stack with functions and its arguments that will be used during the execution of the thread. This initialized thread can be started by simply passing its stack pointer to a thread switching routine. This is the same way a suspended thread is restarted. The context switching works by invoking the QT switching primitive from the new thread's stack and a helper function is executed. This helper function tidies the suspension of the old thread, after which the new thread begins its execution. A detailed description of the context switching primitive is available at [35]. Examples of thread allocation and context switching routines are also provided in [35].

2. QuickThread Client package in SystemC

SystemC has two coroutine packages implemented with QuickThreads. The first is a Unix/Linux variant package and the other is a Microsoft Windows package. We limit our discussion to only the Unix/Linux variant client package. In addition, we only limit our discussion to some class definitions and some specific member functions that we believe are important for the reader to attain a basic idea of the threading coroutine package in SystemC.

Figure A.1 shows a class diagram of the classes used in packaging QuickThreads in SystemC. The implementation of QuickThreads is performed in the *sc_cor_qt* and *sc_cor_pkg_qt* classes. The *sc_cor_qt* class implements the data structure for the stack used in a thread. The class definition is shown in Listing A.1.

The data members of this class define the stack size through *m_stack_size*, a *void* pointer to the stack with *m_stack* and the stack pointer for the thread *m_sp*. This class only holds a pointer to the stack and does not create it. The creating object is *sc_cor_pkg_qt* and hence there is a pointer of that type defined by *m_pkg*. Class *sc_cor_qt* also inherits from an abstract class called *sc_cor*. The only implementation

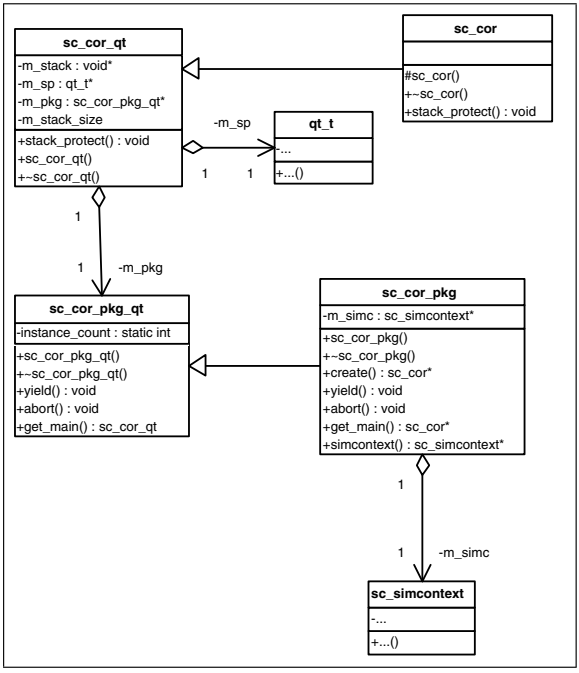


Figure A.1. Class Diagram for some of the Coroutine classes

of *sc_cor_qt* are the constructor, virtual destructor and the *stack_protect* function. The constructor simply initializes all the data members to *NULL* and the destructor frees up the memory if *m_stack* is allocated. The *stack_protect* member function however, is responsible for allocating a stack region and protecting it with appropriate privilege.

The class responsible for creating the thread coroutine is *sc_cor_pkg_qt*, which inherits from an abstract class *sc_cor_pkg*. The class definition of *sc_cor_pkg_qt* is shown in Listing A.2.

There is only one private data member in this class, which holds a *static* integer variable called *instance_count*. This variable is used to ensure that there is only one instantiation of the *sc_cor_pkg_qt* class further enforced by the file static variables *static sc_cor_qt main_cor* and *static sc_cor_qt* curr_cor*. These file static variables are instantiated in the implementation file *sc_cor_qt.cpp*. The coroutine package follows a singleton pattern [19].

The constructor of this class shown in Listing A.3 takes in a pointer to the current simulation context and sets it in the abstract class *sc_cor_pkg* and assigns the current coroutine to the address of the *main_cor* object. No other instantiations are allowed of this class.

The *create(...)* function takes in the stack size, a function and an argument list as mentioned earlier in our discussion of QuickThreads. In this function, the stack size is set in the *sc_cor_qt* object, the stack is allocated memory and the stack pointer is initialized along with passing of the function and arguments. At the end of the *create(...)* function, the thread created is returned as a pointer to *sc_cor* object. The coroutine classes employ the use of keywords such as *SCAST* and *RCAST*. These are

Listing A.1. class sc_cor_qt

```
class sc_cor_qt
: public sc_cor
{
public:

    // constructor
    sc_cor_qt()
  : m_stack_size( 0 ), m_stack( 0 ), m_sp( 0 ), m_pkg( 0 )
  {}

    // destructor
    virtual ~sc_cor_qt()
  { if( m_stack != 0 ) { delete[] (char*) m_stack; } }

    // switch stack protection on/off
    virtual void stack_protect( bool enable );

public:

    size_t          m_stack_size;  // stack size
    void*           m_stack;       // stack
    qt_t*           m_sp;          // stack pointer

    sc_cor_pkg_qt* m_pkg;          // the creating coroutine
        package

private:

    // disabled
    sc_cor_qt( const sc_cor_qt& );
    sc_cor_qt& operator = ( const sc_cor_qt& );
};
```

defined in *sc_iostream.h* as short forms for *static_cast* and *reinterpret_cast*, perhaps simply for easier use.

The *yield(...)* function takes in a pointer to *sc_cor* to indicate the next coroutine to be executed. The current coroutine to be executed is moved to a casted pointer to *sc_cor_qt* and the QuickThread primitive for blocking is called via the *QT_BLOCK(...)* function. The arguments passed to the QT blocking primitive are the helper function, the old coroutine, and the new coroutine. The helper function saves the stack pointer onto the old thread's stack and resumes execution to the new thread using preswitch.

The *abort(...)* member function is responsible for causing the threads to terminate (die) in a similar fashion and the *get_main()* function returns the main coroutine for the simulation to allow continuation of the original simulation context. We do not discuss the abstract classes that are used to interface with the QT coroutine package and continue to discuss how SystemC threads make invocations to the functions described in this section.

We have described some of the classes and some of the main functions of those classes that interact with the QuickThread core package. What is relevant to most SystemC users and more so to developers is how SystemC integrates these coroutine packages with their simulation environment. We describe how the Discrete-Event simulation kernel incorporates thread processes. We step through the code in giving details on how threads are instantiated (up to the coroutine client package calls) and how they are executed.

Listing A.2. class sc_cor_pkg_qt

```
class sc_cor_pkg_qt
: public sc_cor_pkg
{
public:

    // constructor
    sc_cor_pkg_qt( sc_simcontext* simc );

    // destructor
    virtual ~sc_cor_pkg_qt();

    // create a new coroutine
    virtual sc_cor* create( size_t stack_size, sc_cor_fn* fn,
        void* arg );

    // yield to the next coroutine
    virtual void yield( sc_cor* next_cor );

    // abort the current coroutine (and resume the next
        coroutine)
    virtual void abort( sc_cor* next_cor );

    // get the main coroutine
    virtual sc_cor* get_main();

private:

    static int instance_count;

private:

    // disabled
    sc_cor_pkg_qt();
    sc_cor_pkg_qt( const sc_cor_pkg_qt& );
    sc_cor_pkg_qt& operator = ( const sc_cor_pkg_qt& );
};
```

Listing A.3. Constructor from class sc_cor_pkg_qt

```
sc_cor_pkg_qt::sc_cor_pkg_qt( sc_simcontext* simc )
: sc_cor_pkg( simc )
{
  if( ++ instance_count == 1 ) {
    // initialize the current coroutine
    assert( curr_cor == 0 );
    curr_cor = &main_cor;
  }
}
```

The initialization and execution of a thread process starts at the definition of *SC_THREAD(...)* that is defined in *sc_module.h* as a C macro. This macro invokes another macro called *declare_thread_process(...)* that accepts the sensitivity list and actually creates the thread process itself by calling a *register_thread_process(...)* function defined in *sc_simcontext* class. Listing A.6 shows the constructor being called for the *sc_thread_process* class.

Listing A.4. create(...) member function from class sc_cor_pkg_qt

```
sc_cor*
sc_cor_pkg_qt::create( size_t stack_size, sc_cor_fn* fn, void*
    arg )
{
    sc_cor_qt* cor = new sc_cor_qt;
    cor->m_pkg = this;
    cor->m_stack_size = stack_size;
    cor->m_stack = new char[cor->m_stack_size];
    void* sto = stack_align( cor->m_stack, QT_STKALIGN, &cor->
        m_stack_size );
    cor->m_sp = QT_SP( sto, cor->m_stack_size - QT_STKALIGN );
    cor->m_sp = QT_ARGS( cor->m_sp, arg, cor, (qt_userf_t*) fn,
       sc_cor_qt_wrapper );
    return cor;
}
```

Listing A.5. yield(...) member function from class sc_cor_pkg_qt

```
void
sc_cor_pkg_qt::yield( sc_cor* next_cor )
{
    sc_cor_qt* new_cor = SCAST<sc_cor_qt*>( next_cor );
    sc_cor_qt* old_cor = curr_cor;
    curr_cor = new_cor;
    QT_BLOCK( sc_cor_qt_yieldhelp, old_cor, 0, new_cor->m_sp );
}
```

Listing A.6. register_thread_process(...) from class sc_simcontext

```
sc_thread_handle
sc_simcontext::register_thread_process( const char* name,
           SC_ENTRY_FUNC entry_fn,
           sc_module* module )
{
    sc_thread_handle handle = new sc_thread_process( name,
                  entry_fn,
                  module );
    m_process_table->push_back( handle );
    set_curr_proc( handle );
    return handle;
}
```

As it can be seen, a thread process is an object of class *sc_thread_process*. The constructor of *sc_thread_process* sets the default size of the stack defined by *SC_DEFAULT_STACK_SIZE* and the coroutine of that thread (a pointer to *sc_cor*) is initialized to NULL. The newly created thread identified by the handle is pushed onto the process list *m_process_table* for later use. All *SC_THREAD()* processes are pushed onto this process table along with all other process types in SystemC.

The next use of the thread is when simulation is started by *sc_start(...)*. The implementation of this function invokes *sc_get_curr_simcontext()* which creates an instance of *sc_simcontext* and returns a pointer to what would be the Discrete-Event kernel or an object of that type. The *simulate(...)* function is invoked to perform

the simulation. Within this function the initialization of all processes is performed. We extract the segment responsible for thread processes only and display it in Listing A.7.

Listing A.7. Segments of *initialize(...)* from sc_simcontext class

```
1 void
2 sc_simcontext::initialize( bool no_crunch )
3 {
4     // Some code here ...
5
6     // instantiate the coroutine package
7 #ifndef WIN32
8     m_cor_pkg = new sc_cor_pkg_qt( this );
9 #else
10    m_cor_pkg = new sc_cor_pkg_fiber( this );
11 #endif
12    m_cor = m_cor_pkg->get_main();
13
14    // prepare all thread processes for simulation
15    const sc_thread_vec& thread_vec = m_process_table->
          thread_vec();
16    for( int i = thread_vec.size() - 1; i >= 0; -- i ) {
17  thread_vec[i]->prepare_for_simulation();
18    }
19
20
21  // Some code here ...
22
23  // make all thread processes runnable
24
25  size = thread_vec.size();
26  for( int i = 0; i < size; ++ i ) {
27  sc_thread_handle thread_h = thread_vec[i];
28    if( thread_h->do_initialize() ) {
29      push_runnable_thread( thread_h );
30    }
31  }
32
33  // Some code here ...
34
35 }
```

During *initialize(...)*, the constructor of *sc_cor_pkg_qt* is invoked by setting the simulation context as the current *sc_simcontext* object. This is followed by every thread being prepared for simulation by calling the *prepare_for_simulation()* function from the *sc_thread_process* class. This is when the *create(...)* function from the *sc_cor_pkg_qt* is invoked to create the thread, after which the returned pointer to *m_cor* forces a protection on the stack by invoking the *stack_protect(...)* function on that particular coroutine as shown in Listing A.8.

The function passed to the stack initialization is the *sc_thread_cor_fn* that is responsible for calling *execute()* from the base class *sc_process_b*, which mainly executes the entry function of a module. The preparation of the threads allocates stack regions to each thread and initializes the thread stack space with the function and arguments. Once this is done, the thread can be executed. SystemC scheduler dictates that all processes that are not marked by *dont_initialize()* are to be executed during initialization and hence all processes are made runnable by pushing all runnable threads on to the runnable queue. This takes us to the function that performs the Evaluate-Update

Listing A.8. *prepare_for_simulation(...)* from class *sc_thread_process*

```
void
sc_thread_process::prepare_for_simulation()
{
    m_cor = simcontext()->cor_pkg()->create( m_stack_size,
                sc_thread_cor_fn, this );
    m_cor->stack_protect( true );
}
```

paradigm, *crunch()*. We focus mainly on the invocation of *yield(...)* as shown in Listing A.9.

Listing A.9. Use of *yield(...)* in *crunch()*

```
void  sc_simcontext::crunch()
{
  // Some code here ...

    while( true ) {

  // EVALUATE PHASE

  while( true ) {

  // execute method processes

  // Some code here ...

  // execute (c)thread processes

  sc_thread_handle thread_h = pop_runnable_thread();
  while( thread_h != 0 && ! thread_h->ready_to_run() ) {
    thread_h = pop_runnable_thread();
  }

  if( thread_h != 0 ) {
    m_cor_pkg->yield( thread_h->m_cor );
  }

  if( m_error ) {
    return;
  }

  // Some code here ...

    m_runnable->toggle();
  }

  // UPDATE PHASE

  // Some code here ...
}
```

All threads that are on the runnable lists are executed. This raises many questions as to how the simulation proceeds from one thread to the next. For example, suppose a thread switches to the current context and executes. Then, how does another thread after the completion of the current executing thread get scheduled for execution. We

believe that one of the important understandings is how this simulation proceeds to the next thread and how a *wait()* suspension call makes the current executing process suspend **and** switch another process in for execution. The implementation is interesting and behaves differently when there are suspension calls, when there are none, and when the entry function for a thread does not contain an infinite loop. The implementation provides explanation to the experienced behaviors in these circumstances. Let us first study the function shown in Listing A.10. *sc_switch_thread(...)* takes in a pointer to the simulation context as an argument. This function invokes the *yield(...)* function with the argument being the return object of a *next_cor()* function implemented in the *sc_simcontext* class, serving the purpose of retrieving the next runnable coroutine. This *next_cor()* function is shown in Listing A.11.

Listing A.10. *sc_switch_thread(...)* function

```
1 inline void
2 sc_switch_thread( sc_simcontext* simc )
3 {
4     simc->cor_pkg()->yield( simc->next_cor() );
5 }
```

Listing A.11. next_cor() from class sc_simcontext

```
1 sc_cor*
2 sc_simcontext::next_cor()
3 {
4   if( m_error ) {
5     return m_cor;
6   }
7
8   sc_thread_handle thread_h = pop_runnable_thread();
9   while( thread_h != 0 && ! thread_h->ready_to_run() ) {
10    thread_h = pop_runnable_thread();
11  }
12
13  if( thread_h != 0 ) {
14    return thread_h->m_cor;
15  } else {
16    return m_cor;
17  }
18 }
```

This *next_cor()* function returns the coroutine to the next thread if one is available on the runnable queues, otherwise it returns the main coroutine, *m_cor* that returns to the main simulation context. This implies that one thread is somewhat responsible for invoking another thread until the simulation has no more threads to execute at which it returns to the main coroutine.

Suspension calls such as *wait(...)* simply block the current executing thread by switching to another thread using the *sc_switch_thread(...)*, leaving the QuickThread core package to be responsible for saving the state of the old thread. This is how the *main_cor* (main coroutine) is saved as well, when another thread process is invoked, the current executing process is suspended. Therefore, none of these threads are terminated in a normal execution until the end of simulation. The event notifications

for resumption of threads puts the thread process handle on the runnable queue so that the main coroutine can execute the threads. The threads generally terminate only at the end of simulation.

Suppose the user has an *SC_THREAD()* process and no infinite loop implemented in the entry function. Then, the thread will execute only once (depending on the number of suspension points in the entry function) and stall. Suppose, a user has defined an entry function with an infinite loop but no suspension points and without the use of an *sc_signal* or channels that generate *sc_event*s. The expected behavior is an infinite execution of that one process once it is scheduled. The implementation details provide some understanding as to why an infinite loop is required for a thread process in SystemC to prevent the thread from aborting and dying. It explains why all threads also have at least one suspension point to avoid continuous execution of that single thread.

Our efforts in heterogeneity question the current implementation of QuickThreads in SystemC. The concern is not with the QuickThread packaging, but more so the client package interaction with the simulation kernel. The static instance of the *sc_cor_qt* object makes it difficult to implement heterogeneity and hierarchy in SystemC. We believe that every Model Of Computation requiring threads must be able to cleanly communicate to the existing Discrete-Event kernel without raising concerns about threading implementations. We are currently investigating this alteration. However, we believe that this short introduction to QuickThread implementation in SystemC is valuable for kernel designers in understanding the coroutine implementation.

Appendix B
Autoconf and Automake

Autoconf and Automake are tools provided by GNU for making scripts that configure source code packages. Autoconf is responsible for creating a configuration script with information regarding the operating system. This allows for adaptation to different UNIX-variant systems without much intervention from the code developers or the user. Automake is also a GNU tool that requires Autoconf to generate the configuration script which Automake utilizes to create `Makefile.in` files for the code package. Detailed information on the usage and purpose of Autoconf and Automake is available at [21, 22]. Our purpose in this section is to describe how to add additional files to SystemC, such that the library created after compilation will incorporate the additional classes introduced in the added files.

We explain this procedure via example. It is important to save a copy of the directory that contains the QuickThread package. We do not elaborate on the problem except that updating the configuration and `Makefile`s causes a slight disturbance with the QuickThread package. Suppose we wish to add the CSP kernel that has all class definitions in a file called *sc_csp.h* and its respective implementation in *sc_csp.cpp*. We define our source untarred in a directory called `systemc-2.0.1-H/`. The following steps are needed to add the CSP kernel to SystemC (this is specific for version 2.0.1).

1 Copy the CSP kernel source files into the kernel directory `systemc-2.0.1-H/src/systemc/kernel/`

2 Edit the `Makefile.am` in `systemc-2.0.1-H/src/systemc/kernel/` to add *sc_csp.h* under `H_FILES` and *sc_csp.cpp* under `CXX_FILES` in a similar fashion to the existing source.

3 Save the `Makefile.am` and move to `systemc-2.0.1-H/src/` where the `systemc.h` exists.

4 Edit the `systemc.h` file and *#include* the header file for the CSP kernel. For example, add a line `#include "systemc/kernel/sc_csp.h"`.

5 Save `systemc.h` and move to `systemc-2.0.1-H/`.

6 Run `aclocal`.

7 Run `autoconf`. If there are problems due to version discrepancies with Autoconf then run `autoreconf -i`.

8 Run `automake`

9 Begin compiling `systemC-2.0.1-H` as specified in the installation guidelines. Updating of the `Makefile.in`s causes a compile error when trying to compile the QuickThread package. To rectify this, simply remove the existing QuickThread directory and replace it with the backup.

Users of the installation from `systemc-2.0.1-H/` will have access to classes included in *sc_csp.h*. This is a solution to keeping most of the additional source code separate from existing implementation, but avoid changes in original source files is a very difficult task. Hence, we maintain the idea that the newly implemented classes must remain detached with their data structures in a separate file and make necessary changes to original SystemC source files.