Functional Programming and Metamodeling frameworks for System Design

Deepak Abraham Mathaikutty

Thesis submitted to the Faculty of
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Sandeep K. Shukla, Chair
Dong Ha, Member
Michael Hsiao, Member

May 12, 2005
Blacksburg, Virginia

Keywords: Functional Languages, Functional Programming, Metamodeling, Metamodel, Domain, Interoperable Modeling Language, Heterogeneous System Design

Functional Programming and Metamodeling frameworks for System Design

Deepak Abraham Mathaikutty

Abstract

System-on-Chip (SoC) and other complex distributed hardware/software systems contain heterogeneous components whose behavior are best captured by different models of computations (MoCs). As a result, any system design framework for such systems requires the capability to express heterogeneous MoCs. Although a number of system level design languages (SLDL)s and frameworks have proliferated over the last few years, most of them are lacking in multiple ways. Some of the SLDLs and system design frameworks we have worked with are SpecC, Ptolemy II, SystemC-H, etc. From our analysis of these, we identify their following shortcomings: First, their dependence on specific programming language artifacts (Java or C/C++) make them less amenable to formal analysis. Second, the refinement strategies proposed in the design flows based on these languages lack formal semantics underpinnings making it difficult to prove that refinements preserve correctness, and third, none of the available SLDLs are easily **customizable** by users. In our work, we address these problems as follows: To alleviate the first problem, we follow Axel Jantsch's paradigm of function-based semantic definitions of MoCs and formulate a functional programming framework called **SML-Sys**. We illustrate through a number of examples how to model heterogenous computing systems using **SML-Sys**. Our framework provides for formal reasoning due to its formal semantic underpinning inherited from SML's precise denotational semantics. To handle the second problem and apply refinement strategies at a higher-level, we propose a refinement methodology and provide a semantics preserving transformation library within

our framework. To address the third shortcoming, we have developed **EWD**, which allows users to customize MoC-specific visual modeling syntax defined as a metamodel. EWD is developed using a metamodeling framework GME (Generic Modeling Environment). It allows for automatic design-time syntactic and semantic checks on the models for conformance to their metamodel. Modeling in EWD facilitates saving the model in an XML-based interoperability language (IML) we defined for this purpose. The IML format is in turn automatically translated into Standard ML, or Haskell models. These may then be executed and analyzed either by our existing model analysis tools **SML-Sys**, or the ForSyDe environment. We also generate SMV-based template from the XML representation to obtain verification models.

Acknowledgements

I am extremely grateful to my advisor, Dr. Sandeep K. Shukla for advising and supporting me throughout this work. We acknowledge the support of ICTAS and the NSF CAREER grant CCR-0237947 which provided the funding for the work reported in this thesis.

I would like to acknowledge my roommates/colleagues Debayan and Suhaib for motivation and support. I would like to specially thank Hiren for his contribution to the development of **EWD**.

I would like to thank my friends Gaurav, Animesh, David and Nimal, for their support.

I would also like to thank my family and my sister's family for their support.

Finally, I would like to thank my friends ('lapisblend' & 'kuwaitrulers99') and my girl for being patient and understanding.

Dedication

I dedicate this thesis to ...

Mr. Manattu Abraham Mathaikutty

and

Mrs. Sosamma Mathaikutty

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The technological advances experienced in the last few decades has initiated an increasing trend towards System-on-Chip (SoC) design, where such systems integrate micro-controllers, digital signal processing cores, memories, application specific logics, and reconfigurable hardware in the form of field programmable gate arrays (FPGAs) together with a communication structure and analog-to-digital (A/D) and digital-to-analog (D/A) converters on a single chip. As a result of Moore's law [18], systems are designed with more and more components on a single chip to offer enormous potential and in addition the technological advances exponentially reduce the cost and size of such SoCs. This provides system designers with the opportunity to design large and complex systems while also meeting the ever increasing demands of the consumer. However, the complexity in the design does not only arise from the miniaturization of systems onto a single chip, but it also emerges from its heterogeneous nature, that is from the fact that in complex designs that interact with the real world, different parts are appropriately captured using different models and methodologies. As a result, the gap between the complexity of the design and the engineering efforts required to realize these designs

are increasing drastically. This problem is commonly known as the productivity gap, as shown in Figure 1.1. 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 increase in the productivity gap, industries find it difficult to meet the stringent time-to-market requirement. Efforts towards mitigating the productivity gap has led to the evolution of several design methodologies of which the one we address are System Level Design Languages (SLDL)s, that would handle the design complexity and raise the abstraction level so as to mitigate the productivity gap.

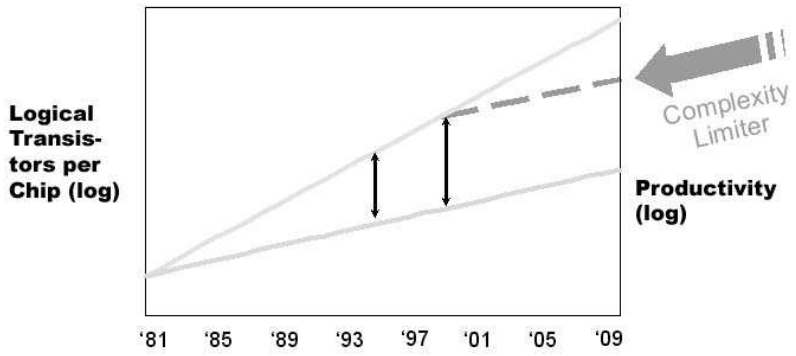


Figure 1.1: Productivity Gap

1.0.1 System Level Design Languages and Frameworks

For a modeling language and framework to be effective and address complex systems, it should start at high levels of abstraction by separating functionality of the system from architecture, communication from computation and promoting the use of formal models and transformations in system design to leverage formal verification. This

led to the development of a number of SLDLs, which provide users with a collection of libraries of data types and kernels, and components accessible through either graphical or programmatic means to model systems and simulate the behaviors. Examples of recently introduced system level modeling languages are SpecC, SystemC and SystemVerilog [34, 21, 30]. Hardware modeling languages like VHDL [37] and Verilog [36] allow designers to create models through programmatic means for RTL designs. SystemC [21] is a library of C++ classes that facilitates objects oriented modeling. This allows modeling at a higher level of abstraction than RTL models making it appropriate for system level designs. A Java-based embedded systems modeling language developed by U.C. Berkeley is Ptolemy II [24] that allows for modeling through a graphical interface.

Most system models for SoCs are heterogeneous in nature, and encompass multiple models of computation in its different components. As a result, we need a framework that provides a way to express heterogeneous models of computation for modeling SoCs. Most SLDLs mentioned above encompass a single model of computation, for instance the languages VHDL, Verilog and SystemC share the same discrete time, event driven computational model. However, Ptolemy II [24] is a design framework, which is used to model heterogeneous embedded systems. Recently several researchers have been enhancing C++ based libraries such as SystemC for capturing heterogeneity in such models. The outcome SystemC-H [22, 23] allows designers to put together a heterogeneous model without worrying about the target simulation kernel. However, few of these SLDLs are endowed with rigorous formal semantics to make use of any semantic formalism during synthesis or verification. Furthermore, none of these are free of the programming artifacts of the specific language (Java or C++), and hence the computation inherent in the model is obscured by such side issues. Moreover, often the computation and communication aspects between these components models get intertwined. As a result, we need a framework that provides a way to express heterogeneous models of

computation for modeling SoCs that have well-defined formal semantics. To alleviate these problems, we follow Axel Jantsch's paradigm [8] of function-based semantic definitions of models of computation, and implement a modeling framework with multiple models of computation in a functional language Standard ML [17]. It is based on formal semantics and functional paradigms which facilitate the application of formal methods for transformation, synthesis and verification. Defining models of computation as higher order functions provide well-defined modeling semantics and facilitate the integration of models of computation, languages and tools at both the syntactic and semantic level. In this respect it goes well beyond other frameworks such as SystemC-H and Ptolemy II, and it will allow for the development of cross-domain analysis, synthesis and validation tools beyond pure simulation.

Another limitation with the available SLDLs is that they are not easily **customizable** by users, as possible in metamodeling frameworks. Providing the user with customizable modeling syntax helps in reducing the number of errors by catching it earlier in the design. To facilitate such syntax, models of computation are described as metamodels that express the syntactic structure and the semantics that the designer is provided with for modeling.

1.0.2 Model of Computation

A model of computation (MoC) is a description mechanism that defines a set of rules to mimic a particular behavior [9, 12]. It is a mathematical formalism that describes the interaction between components in a system. MoCs describe the following:

- How each component performs internal computation.
- How the components transfer information between them.

- How they relate in terms of concurrency.

The most renowned classification of MoCs in the embedded system design community has been stated in the context of Ptolemy and Ptolemy II projects [24] at the University of California at Berkeley. Ptolemy II is built with multiple MoCs, which include various sequential MoCs such as FSM, Discrete-Time, Continuous Time models as well as models of interacting entities, such as CSP, KPN, interaction models etc. These MoCs are referred to as modeling domains in the context of Ptolemy II.

Another distinguishing classification of MoCs was done by abstracting functionality of complex designs in the context of ForSyDe project [27] in Sweden by Axel Jantsch and his group. This work can be distinguished from Ptolemy group's work as a distinction of the denotational view verses operational view of MoCs. A denotational view of a MoC consist of a set of process constructors and process composition operators with denotational semantics. This view classifies MoCs based on the abstraction of the timing behavior of the model.

The three main MoCs in this view are

1. **Untimed Model of Computation:** This is the most abstracted model of computation in terms of the timing behavior. In other words, we do not worry about the time taken for the computation performed nor the time for the different processes to interact and synchronize.

2. **Synchronous Model of Computation:** The timing aspect is abstracted to cycles of computation in this MoC. It can be further categorized into two sub-categories, The first sub-category follows the *perfect synchrony hypothesis*, in which computation and communication takes zero time and the second category follows the *clocked synchrony hypothesis*, in which a global clock enforces the beginning and end of a

computational cycle. Therefore, every computational cycle takes exactly one clock cycle and communication takes no time.

3. **Timed Model of Computation:** This is the least abstracted model of computation, where the timing aspect of all computation and communication is exactly preserved.

The operational view of MoC provided by Ptolemy II describe, exactly how the computation takes place and how the communication proceeds between the different processes and across MoCs in the true operational sense. Some of the MoCs in this view are:

1. **Discrete Event:** In this MoC, all events are associated with a time instant and the time is represented as a discrete set. There is a global event queue and the events describe the combinational behavior of the system being modeled. The computation is characterized into cycles, whereas the communication aspect is not distinctly separated out in this domain.

2. **Finite State Machine:** This MoC is built on states and state transitions, where states describe the computation and communication occurs through transitions between the states.

3. **Continuous Time:** In this MoC, the computation process involves modeling differential equations over the continuous domain of real-numbers.

4. **Communicating Sequential Processes:** This MoC differs from all the other MoCs in the way communication proceeds. Processes execute concurrently and the *rendezvous* communication protocol dictates the transfer of data which only occurs when both communicating processes are ready to communicate.

Ptolemy II has multiple modeling domains that facilitate designers to model in a truly heterogeneous manner, therefore Ptolemy II is a **multi-MoC framework**. Two important characteristics of a multi-MoC framework are fidelity and expressiveness [22]. *Fidelity* is defined as the degree to which a theoretical MoC can be modeled in a framework and *expressiveness* is the availability of a programming language to a user to model systems according to the user's own modeling style. Therefore, a desired quality in any multi-MoC framework should be high fidelity with low expressiveness. The low expressiveness is surmounted with high fidelity that provides the designer with all the common computational models and restricts them to specific process constructors and compositional operators. This reduces modeling errors by the designer to achieve the correct behavior of a model.

MoCs described by functional languages [15, 7, 25, 20] have a clean and simple semantic model, where in a computation is described as a function application thereby providing a more abstract notation for expressing computation. Therefore, modeling in a functional framework has the advantage of being readily susceptible to formal verification.

1.1 Functional Programming in Embedded Systems Design

A functional program is a function that receives the program's input as argument and delivers the program's output as result. A function is free from side-effects, i.e. it has no internal state. This means that the whole functional program is free from side-effects and it is totally deterministic. Functional programs contain implicit parallelism, which is very useful when dealing with embedded system applications, since they typically have a considerable amount of built-in parallelism. Of course it is also possible to parallelize

imperative languages like C++, but it is much more difficult to extract parallelism from programs in such languages, since the flow of control is also expressed by the order of statements. While functional languages fit naturally for data flow applications, they also provide a rich variety of control constructs, making it more suitable for control-dominated applications.

Functional programming is also seen to be highly relevant to the understanding of real-time systems. A computation expressed as a function and its interaction with the outside world being modeled as inputs given to the function, and for these interactions to be infinite in nature, they are modeled using *streams* [11]. This description is well-suited for reactive and interactive systems. The types and type systems [31] in functional languages can be used as a tool for classifying MoCs for real-time applications.

1.2 MoC as a Metamodeling Domain

Configuring MoCs as metamodel requires the designer modeling in that domain to obey the syntactic and semantic constraints of that MoC-specific metamodel. Thinking of MoCs as metamodels allows us to take advantage of the syntactic structure and limited semantics of a metamodel to catch modeling errors at early stages of the design. Furthermore, it is essential to flag modeling errors that are not caught by the compiler of the programming language in which these MoCs are defined. The syntactic structure of an MoC can be expressed using the components (UML classes) provided by the modeling paradigm in GME [35, 10] and semantics are expressed as OCL constraints. The designer instantiates a metamodel that makes available a modeling domain with the respective computational components and composition operators to create a model instance. The actual behavioral aspect of the model is embedded in the attributes associated with these

computational components that can be composed respectively to create the model.

1.3 Main Contributions

Our main contributions in this thesis can be summarized as follows:

1. A multi-MoC modeling framework built on Axel Jantsch's classification of generic MoCs named **SML-Sys** and implemented in Standard ML [17].
2. A semantic-preserving refinement strategy that formulates a design flow targeted towards an optimized synthesis process.
3. A visual modeling environment for multi-MoC modeling of embedded software and hardware systems, named **EWD**, which is built on top of a metamodeling environment GME [10, 35].
   (a) Definition of an interoperability language based on XML, named *IML* that is an abstract extensible syntax to express framework independent MoCs.
   (b) An *xmlTree* structure to facilitate multi-targeted code generation built in C++.
   (c) Translation streams *SmlStream* and *HaskellStream*, to automatically transform models into **SML-Sys** or ForSyDe [38] executables.
   (d) Finally, *SmvStream*, a partial stream to translate the model to the SMV [16] model checker for verification purpose.

1.4 Organization

This thesis is organized as follows:

In Chapter 2, we provide background material on Axel's theory on the generic MoCs, functional programming and semantics, functional design frameworks, refinement methodologies and finally metamodeling and the generic modeling environment.

In Chapter 3, we explain our implementation of the different MoCs in a functional language Standard ML followed by the interfacing of these different MoCs, providing designers with a multi-MoC modeling framework **SML-Sys**. Furthermore, we illustrate the modeling of example like FIR, Sobel and Traffic Light Controller in our framework.

In Chapter 4, we describe our refinement methodology and how it fits into our design flow that start with an abstract functional specification and ends with an efficient implementation specification, that gears up towards a optimized synthesis process.

In Chapter 5, we present **EWD**, A Metamodeling Driven Customizable Multi-MoC System Modeling Environment. We also provide details of the model construction and the multi-targeted code generation phase of the **EWD** design flow.

In Chapter 6, we describe the platform-independent modeling langauge (IML), which is XML-based representation for generic MoC. Furthermore, we explain the parsing phase of the EWD design flow in detail.

Chapter 7 concludes this thesis.

Chapter 2

Background

2.1 Generic MoCs

An MoC is chosen for describing a sub-behavior of a design based on its suitability: compactness of description, fidelity to design style, ability to synthesize and optimize the behavior of an appropriate implementation. We briefly introduce the generic MoCs defined by Axel Jantsch in [8]. These MoCs are built on processes, events and signals. *Events* are the elementary units of information exchanged between processes. *Processes* receive or consume events, and they send or emit events. *Signals* are finite or infinite sequence of events. The activity of processes is divided into *evaluation cycles*. In each evaluation cycle, a process consumes input and emits output. A process partitions its input and output signals into sub-sequences, such that during each evaluation cycle it consumes exactly one sub-sequence of each of its input signals. To relate functions on events to processes, we introduce process constructors, which are parameterizable templates that instantiate processes. We define a number of process constructors: some

with no internal state and others with internal state, some with one input and one output and others with several inputs and outputs. Furthermore, we define process combinators, to construct process networks by composing processes.

An MoC is defined as a set of processes and process networks that are constructed from the given set of process constructors and combinators. We finally categorize MoC based on "how the processes communicate and synchronize" with other processes and, in particular with the "timing information" available to and used by the process.

Definition 2.1.1 *In [8], a model of computation (MoC) is defined as* a 2-tuple framework MoC = (C, O), where C is a set of process constructors, each of which, when given constructor-specific parameters, instantiates a process. O is a set of process composition operators which, when given processes as arguments instantiates a new process.

MoCs are characterized by the duration of their evaluation cycles. The three generic MoC's defined in [8] are: Untimed MoC, Synchronous MoC and Timed MoC.

2.1.1 Preliminary Notations

We introduce the notations used in defining and distinguishing the generic MoCs. A process communicates with another process by writing to and reading from signals. The set of values V represents the data communicated over a signal and the set E constitutes of events that are basic elements of a signal containing values. We distinguish between three different kinds of events. Firstly, untimed events that are values denoted by $\dot{E} = V$. Secondly, synchronous events denoted by $\bar{E}$, which also includes a pseudovalue $\sqcup$, (the absence of an event), hence $\bar{E} = V \cup \{\sqcup\}$ and finally timed events denoted by $\hat{E}$ that are identical to synchronous events, $\bar{E} = \hat{E}$. Therefore E = $\dot{E} \cup \bar{E} \cup \hat{E}$ and $e \in E$ denotes any

kind of event. Signals are ordered sequence of events such that e_i denotes the i^{th} event in a signal. We use $\dot{S}, \bar{S}$ and $\hat{S}$ to denotes the untimed, synchronous and timed signal sets, such that S = $\dot{S} \cup \bar{S} \cup \hat{S}$, and $\dot{s}, \bar{s}$ and $\hat{s}$ designate individual untimed, synchronous and timed signals respectively. Processes are defined as functions on signals p: $S \rightarrow S$ that is a mapping between signal sets. Furthermore, they are allowed to have internal state such that for the same given input signal they react differently at different time instances.

The Untimed Model of Computation

The untimed MoC is characterized by the way its processes communicate and synchronize with other processes without the notion of time such that only the order of events are relevant. Thus, the untimed MoC operates on the *causality abstraction*.

The Synchronous Model of Computation

The Synchronous MoC divides the time line into intervals. Every computation within an interval occurs at the same time, but the intervals are totally ordered along the time line. In synchronous MoCs the evaluation cycle of processes lasts exactly one time interval. We further categorize synchronous MoCs based on whether the output event of a process occurs in the same time interval as the corresponding input event or whether every process incurs a delay from an input event to an output event.

Perfect Synchronous Model of Computation: This MoC is built on the basis of the *perfect synchrony hypothesis*, where the output events of a process occur in the same time interval as the corresponding input events. Moreover they are instantaneously distributed in the entire system and are available to all other processes in the same time interval. Receiving

processes consume input events and emit output events in the same time interval.

Definition 2.1.2 ***Perfect synchrony hypothesis:*** *Neither computation nor communication takes time.*

The Clocked Synchronous Model of Computation: This MoC is based on the *clocked synchronous hypothesis.* It differs from the perfectly synchronous MoC in that every process incurs a delay from an input event to an output event. The delay is equivalent to the duration of an evaluation cycle. For clocked synchronous processes, we introduce a delay function $\triangle$, which delays each input by one clock cycle.

Definition 2.1.3 ***Clocked synchronous hypothesis:*** *There is a global clock signal controlling the start of each computation in the system. Communication takes no time, and computation takes one clock cycle.*

Timed Model of Computation

This MoC is a generalization of the synchronous MoC. Timing information is conveyed on the signals by transmitting absent events at regular time intervals. In this way, processes always know when a particular event has occurred and when no event has occurred. It differs from the synchronous MoC on two accounts, the granularity of the timing structure is much finer and a process can consume and emit any number of events during one evaluation cycle.

2.2 Implementing MoC frameworks in Standard ML

SML provides compound datatypes like lists, tuples, records and allows abstract datatypes. We implement finite signals as generic lists and infinite signals as delayed function applications. The notion of delayed function application allows us to implement lazy evaluated semantics [7], and avoid the overflow exception that occurs due to inherent eager evaluated SML semantics. The implementation of finite and infinite signals is shown in Listing 2.1.

Listing 2.1: Definition of signals

```
1 (* Definition of a Finite signal *)
2 datatype 'a signal = nil | :: of 'a * 'a signal
```

We formulate a few signal manipulators shown in Listing 2.2 that can handle sequence of events from finite signals for the different process constructors.

Listing 2.2: The Signal Manipulating functions

```
1  exception Empty_Seq (* Exception defined for empty finite signal *)
2
3  fun take (x::y,0) = []
4  |   take ([],0)   = []
5  |   take (x::y,n) = x::take (y,n-1)
6
7  fun drop ([],_)   = []
8  |   drop (x::y,0)= x::y
9  |   drop (x::y,n) = drop (y,n-1)
10
11 fun tail [] = raise Empty_Seq | tail (x::y) = y
12 fun head [] = raise Empty_Seq | head (x::y) = x
13
14 fun length(s) = (case s of [] => 0 | s => 1 + length(s))
15
16 fun partition ([],_) = []
17 |   partition (_,[]) = []
18 |   partition (v1::v2,x::y) = (if length (x::y) < v1
```

```
19 then [] else [take (x::y,v1)] @ partition (v2,drop (x::y,v1)))
```

The function **take(n, s)** extracts the first n elements of a signal s and function **drop(s, n)** deletes the first n elements of a signal s. The function **head(s)** extracts the first element of signal s and function **tail(s)** deletes the first element from a signal s. Moreover, function **length(s)** returns the length of the signal s. Finally, the function **partition(v, s)** divides the signal into an ordered set of signals, that when concatenated, form the original signal s. Each ordered set is a sub-sequence. The argument v is the length of each partition.

The function *take(n, s)* extracts the first n elements of a signal s and function *drop(n, s)* deletes the first n elements of a signal s. The function *head(s)* extracts the first element of signal s and function *tail(s)* deletes the first element from a signal s. Moreover, function *length(s)* returns the length of the signal s. Finally, we define a partition function $\Psi(v, s)$ that divides the signal into an ordered set of signals, that when concatenated, form the original signal s. The argument v: N $\longrightarrow$ N is the length of each partition.

Definition 2.2.1 *The partition function is defined as:*

$$\Psi(v,s) = \begin{cases} \langle take(s, v(i))\rangle \oplus \Psi(v, drop(s, v(i))) & if\ length(s) \geq v(i) \\ \langle\rangle & otherwise \end{cases}$$

$$where\ s \in S, i \in N$$

The datatype for infinite signal is defined differently than for finite signals. An infinite signal is the delayed evaluation of a function application to a finite stream of the infinite signal. The recursive definition is shown in Listing 2.3.

Listing 2.3: Definition of an infinite signal

```
1 (* Definition of an event-based Infinite Signal *)
```

```
2  datatype 'a signal = nil | const of 'a * (unit -> 'a signal)
3  fun next (k) = const(k, fn() => next(k+1))
```

The *const* defines a delayed function application of *next* function using the unit operator of SML. The *next* function defines the sequence of events for the signal, it takes an initial event to compute the next event of the infinite sequence. In our case, the *next* function maps events to the set of natural numbers.

The functions *head*, *tail*, *drop*, *take*, *partition* are extended to handle infinite signals as shown in Listing 2.4.

Listing 2.4: Infinite Signal Manipulating functions

```
1  (* Exception defined for empty infinite signal *)
2  exception Empty_Inf_Seq
3
4  fun take (nil,n) = raise Empty_Inf_Seq
5  |   take (s,0)   = []
6  |   take (e_const(x,y),n) = x::take (y(),n-1)
7
8  fun drop (nil,_) = raise Empty_Inf_Seq
9  |   drop (s,0)   = s
10 |   drop (e_const (x,y),n) = drop (y(),n-1)
11
12 fun head nil = raise Empty_Inf_Seq | head (e_const (x,y)) = x
13 fun tail nil = raise Empty_Inf_Seq | tail (e_const (x,y)) = y()
14
15 fun partition ([],_)  = []
16 |   partition (_,nil) = raise Empty_Inf_Seq
17 |   partition (v1::v2,e_const (x,y)) =
18 [take (e_const (x,y),v1)] @ partition (v2,drop (e_const (x,y),v1))
```

2.3 Functional Programming and Semantics

Functional programmers restrict themselves facilities which other programmers regard as standard. When using functional languages [7, 25, 20], we do away with notions such as variables and reassignments. This allows us to define programs which can be subjected to analysis much more easily. There is no state corresponding to the global variables of a traditional language or the instances of objects in an object oriented language. Each definition is treated as binding that is active throughout the execution of the program. Reassignment are not permitted which makes functional languages declarative rather than imperative. A declarative language is one which the programmer declares what the problem is and the execution is a low level concern. Functional programs are conducive to formal development methods. In particular, the ability to reason about functional programs improves the effectiveness of formal inspections. We compare functional languages with some of the traditional imperative languages which is used to implement MoC in Table 2.1.

2.3.1 Why Standard ML?

Standard ML (SML) [17] was one amongst the new programming languages developed in the 1980s and was seen as a suitable vehicle for serious systems and applications programming. It offers an excellent ratio of expressiveness to language complexity, and provides competitive efficiency. SML manages to combine safety, security, and robustness with a great deal of flexibility because of its type and module system. We employ SML to implement the generic models of computation for the following reasons:

- SML provides a great deal of expressiveness by its ability to treat functions as

Table 2.1: Comparison between Functioning and Traditional programming

Imperative Programming	**Functional Programming**
Uses informal specification - may be open to interpretation.	Uses logic to state the specification exactly.
Uses appropriate testing strategies to improve reliability	Uses the underlining mathematical model to prove correctness.
Errors are common and difficult to spot and correct.	Errors are uncommon.
Uses structured programming or object oriented techniques for code reuse. Problems can be partitioned into more manageable chunks.	Uses functional programming for reuse code and partitioning problems into easy to use chunks. Furthermore, uses "higher-level" abstractions that are impossible in traditional languages.

first-class values, and its usage of higherorder functions and the availability of imperative constructs which provide great expressive power within a simple and uniform conceptual framework.

- SML provides a highlevel model which makes programming more efficient and more reliable by automating memory management and garbage collection.
- SML does static type checking which detects many errors at evaluation time. Error detection is enhanced by the use of pattern matching and by the exception mechanism.
- SML module system is an organic extension of the underlying polymorphic type system thereby providing separation of interface specification and implementation. These facilities are very effective in structuring large programs and defining generic, reusable software components.

2.3.2 Functional Language-based System Design

Functional languages have been used in other system design research projects. We briefly introduce some of them and show that **SML-Sys** supersedes them in different aspects.

HML [13] is a hardware description language based on SML, which combines strong typing with polymorphism and automatic type inference to express the functionality of the hardware being specified. It is mainly an improvement over VHDL [37] that does a HML-to-VHDL translation targeting a synthesizable subset of VHDL.

A hardware/software co-design description language based on SAFL (Statically Allocated Functional Language) has been presented in [28]. SAFL is a functional language with ML syntax. Furthermore, design descriptions in SAFL are translated to hardware, targeting hierarchical Verilog [36] RTL.

Bluespec [3] is a language for hardware design that borrows its notation, type and package system from an existing general-purpose functional programming language Haskell. It is meant for hardware design and attempts to raise the level of abstraction without compromising the quality of hardware generated. The Bluespec tool-set is based on the SystemVerilog [30] language.

Ruby is a declarative language of relations and functions based on Haskell. [14] describes a hardware/software codesign based on Ruby that expresses hardware functionality as relational descriptions and via a compiler translates this description into VHDL.

The Lava system [2] is an extensible tool to assist hardware designers in designing, verifying and implementing hardware. It is a collection of Haskell modules that exploit functional programming features to provide circuit descriptions. It focuses on structural

representation of hardware as in Ruby and provides a variety of compositional patterns.

ForSyDe [27] is a library-based implementation that provides a computational model for only the synchronous domain with interfaces implemented in Haskell [6, 33]. The ForSyDe design process starts at a higher abstraction level with a synchronous formal specification model. The synthesis process is divided into two phases. In the first phase, the specification model is refined into a more detailed implementation model by the stepwise application of design transformations shown in [38]. The second phase is the mapping of the implementation model onto a given architecture. This phase comprises activities like partitioning, allocation of resources and code generation. Synthesizable VHDL and C is generated for HW and SW implementation, respectively.

2.3.3 Refinement Frameworks

ForSyDe's refinement methodology based on their transformation library defines different refinements that are either semantic preserving or based on design-decisions. The design-decisions introduce low-level implementation details at the different refinement stages that restricts the methodology from automating these refinements.

The SpecC [34, 29] framework describes refinement strategies from behavioral specifications to architectural model through mapping of behaviors onto processing elements and synthesizing communication primitives necessary for the mapping. However, the behavioral models in such an approach have computational threads and channel based communication between hierarchy of threads. The architectural model amounts to the mapping of such threads and communication mechanisms onto processors and buses. There are two fundamental problems with these approaches (i) the lack of mathematically sound operational semantics which preserve the congruence relations on mapping

across architectural elements, rendering the theoretical proof of correctness weak at its best, (ii) the refinement steps preserve partially ordered trace equivalence showing that the behavioral and architectural models are both at much lower abstraction level than required for system design.

A formal refinement-checking methodology for system-level design based on a polychronous MoC of the multi-clocked synchronous formalism is called SIGNAL [32], implemented in the POLYCHRONY workbench. They also demonstrate the effectiveness of this approach by an experimental case study of a Even-Parity Checker example. However their methodology has a distinct design language (SpecC) and a distinct refinement language (SIGNAL) which gives rise to a disconnect.

2.4 Why SML-Sys Framework?

SystemC-H and Ptolemy are built on imperative languages like C++ and Java that obscure the computation inherent in the model due to the side-effects of language-specific artifacts. We alleviate these problems in **SML-Sys**, which is based on formal semantics and functional paradigms that have an underlining mathematical model. Furthermore, Ptolemy presents an operational view of MoCs, which include various sequential MoCs such as FSM, Discrete-Time, Continuous Time models as well as models of interacting entities, such as CSP, etc. However, **SML-Sys** framework based on [8], describes MoCs denotationally, which consist of a set of process constructors and process composition operators with denotational semantics. It abstracts out information regarding (i) how the computation is performed, (ii) how the communication between the different processes in a model proceeds, and (iii) timing behavior of the model to various levels.

SML-Sys has a high modeling fidelity, since it is a multi-MoC modeling framework

based on the generic definition in [8], which is an extension of the ForSyDe methodology. ForSyDe has a low modeling fidelity, since it is based on a single computational model. ForSyDe illustrates the formulation of function-based semantics of MoCs by defining the synchronous MoC. We take this a few steps further and formulate the untimed, clocked synchronous and timed MoCs. ForSyDe's computational model is based on the synchrony assumption and is best suited for applications amenable to synchrony, which limits it. In our **SML-Sys** framework, we have formulated an untimed MoC that does express dataflow models and state machines with ease, a clocked synchronous MoC to model digital hardware and finally a timed MoC to model real-time requirements like timing analysis to compute clock cycle width. Furthermore, we integrate these MoCs, making **SML-Sys** a multi-MoC modeling framework. The integration is formulated by defining the interfaces between similar and different MoCs. These interfaces facilitate inter-domain modeling by handling the timing aspects of the different integrated domains.

The hardware/software co-design based on SAFL and Bluespec are similar to **SML-Sys** and ForSyDe with their intent in raising the level of abstraction by starting with a functional specification than a behavioral one. However, it is limited to a single modeling domain. Bluespec, HML, Lava and Ruby differ from **SML-Sys** in that they perform hardware modeling and design at lower levels of abstraction. Furthermore, **SML-Sys** executable models, can be translated to VHDL/Verilog based RTL descriptions, which has been demonstrated by ForSyDe.

SML-Sys's design-flow starts at a functional specification and is refined into a communication specification using semantic preserving refinements, and furthermore into an efficient implementation specification. ForSyDe differs from **SML-Sys** in that their communication refinements are based on design choices, which does provide them more flexibility but imposes the restriction that the design-flow cannot be automated.

SML-Sys performs semantic preserving-based refinement, which is also the case for the SAFL-based framework. is similar to the **SML-Sys** framework since they perform semantic preserving-based refinements.

The problems encountered in SpecC are eliminated in **SML-Sys**, since our approach targets a higher level of abstraction, where the initial specification is purely functional (separating computation from communication) and does not include behavioral aspects such as computational threads and channel based communication between hierarchy of threads.

SML-Sys is a library-based functional framework for multi-MoC modeling for heterogeneous system design. The design-flow in **SML-Sys** starts with writing a functional specification that is transformed into an optimized implementation specification through well-defined semantics preserving refinements.

2.5 MetaModeling & MetaModeling Environment

Metamodeling [19] is defining a *metamodel* to describe a modeling system. Metamodels are formalized descriptions of the objects, relationships, and characteristics required in a particular domain-specific MIPS environment (DSME). In MIPS (model-integration program synthesis), models are created to capture various aspects of a domain-specific system's desired characteristics. Model interpreters are used to translate these models for use in the system's execution environment, either as stand-alone applications or in conjunction with code libraries and some form of middleware (e.g. CORBA, the MultiGraph kernel (MGK), POSIX, etc.) When changes in the overall system require new application programs, the models are updated to reflect these changes, and the applications are regenerated automatically from the models. Metamodeling can be used

to define a metamodel that is a DSME itself, providing a modeling environment, which in a manner is similar to defining a metamodel that is a domain-specific application. Such a DSME is called a *meta-metamodel* and an environment that facilitates such metamodeling is called a *metamodeling environment*.

We provide a few definitions below to avoid confusion:

1. A **model** is an abstract representation of any system.
2. A **modeling environment** is a modeling paradigm for creating, analyzing, and translating domain-specific models.
3. A **metamodel** formally defines the syntax and semantics of a particular domain-specific modeling environment.
4. A **metamodeling environment** is a framework for creating, validating, and translating metamodels.
5. A **meta-metamodel** formally defines the syntax and semantics of a given metamodeling environment.

2.5.1 Generic Modeling Environment

Generic Modeling Environment (GME) developed at the Institute for Software Integrated Systems at Vanderbilt University is a Windows-based, generic, configurable toolkit for creating domain-specific modeling and program synthesis environments.

1. It is used primarily for model-building. The models take the form of graphical, multi-aspect, attributed entity-relationship diagrams. The dynamic semantics of

a model is not the concern of GME that is determined later during the model interpretation process.

2. It supports various techniques for building large-scale, complex models. The techniques include: hierarchy, multiple aspects, sets, references, and explicit constraints.

3. It contains one or more integrated model interpreters that perform translation and analysis of models currently under development.

The configuration is accomplished through *metamodels* specifying the *modeling paradigm* of the application domain as shown in Figure 2.1. The modeling paradigm contains all the syntactic, semantic, and presentation information regarding the domain. It also provides an interface to describe which concepts will be used to construct models, what relationships may exist amongst these concepts, how the concepts may be organized and viewed by the modeler and rules governing the construction of the model. The modeling paradigm (MP) defines the family of models that can be created using the resultant modeling environment created by configuring a *metamodel* using the GME *metametamodel*. The metamodels specifying the modeling paradigm are used to automatically generate the target domain-specific environment. The generated domain-specific environment(MP) is then used to build domain models that are stored in a model database. These models are used to automatically generate the applications or to synthesize input to different Commercial Off-The Shelf (COTS) analysis tools. This process is called *model interpretation*. The metamodeling paradigm is based on the *Unified Modeling Language* (UML). The syntactic definitions are modeled using pure UML class diagrams and the static semantics are specified with constraints using the *Object Constraint Language* (OCL). GME also provides bidirectional XML access for both model and metamodel information.

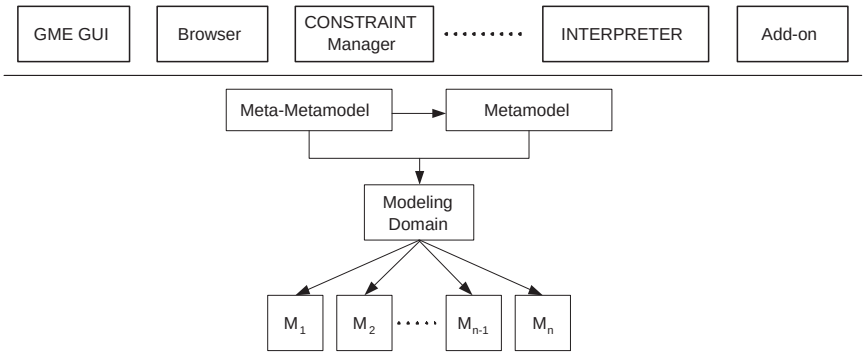


Figure 2.1: GME Architecture

The vocabulary of the domain-specific languages implemented by different GME configurations is based on a set of generic concepts built into GME itself. These include: hierarchy, multiple aspects, and explicit constraints. *Folders*, *Models*, *Atoms*, *Connections*, *Roles*, *Constraints* and *Aspects* are the main elements in defining a modeling domain. The metamodel is saved as a *Project* which contains a set of *Folders*. *Folders* are containers that help organize models. *Models*, *Atoms*, and *Connections*, are first class objects (FCO)s. *Atoms* are the elementary objects that cannot contain parts and have a predefined set of attributes. *Models* are compound objects with an inner structure. GME's modeling concept *Aspects* provides visibility control. Every Model has a predefined set of *Aspects*. Each part can be visible or hidden in an *Aspect*. The simplest way to express a relationship between two objects in GME is with a *Connection*. *Connections* can be directed or undirected. *Connections* can have *Attributes* themselves. In order to make a *Connection* between two objects they must have the same parent in the containment hierarchy (and they also must be visible in the same Aspect) *Connections* can further be restricted by explicit constraints specifying their multiplicity. The kinds of *Attributes* available are text, integer, double, boolean and enumerated. The modeling language is made up of instances of these concepts.

When a particular metamodel is created, it becomes a type (class). It can be subtyped and instantiated as many times as the user wishes. This makes it possible to create libraries of type models that can be used in multiple applications in the given domain. Only attribute values of model instances can be modified. The model instances have to be in syntactic conformance with the metamodel and should also abide to all the constraints imposed on the metamodel.

Chapter 3

Implementing MoCs with SML

In this chapter, we describe the SML-based implementation of the generic MoCs in the **SML-Sys** framework. Notations used to formalize the components of a generic MoC is discussed in Chapter 2. We discuss the implementation of the untimed, synchronous and timed MoC and illustrate through a number of examples how to model interesting systems using a framework with underlining denotational semantics, bereft of any linguistic artifacts. Finally, we conclude this chapter by formulating the interfaces between MoCs to facilitate inter-domain modeling.

3.1 Untimed Model of Computation (UMoC)

UMoC adopts the simplest timing model that follows the causality abstraction. Processes, modeled as state machines, are connected to each other via signals. Signals transport data values from a sending process to a receiving process. The data values do not carry time information, but the signals preserve the order of emission.

3.1.1 Process Constructors

In the UMoC, process constructors are templates on processes, which are implemented as higher-order functions that take functions on events as argument and instantiate processes. We implement a set of process constructors that are used to define computational blocks that are either complex processes or process networks. We suffix it with *U* to designate it to the UMoC.

map-based process constructor: It is a stateless constructor that creates a process, which takes an input signal and generates an output signal. It takes a constant 'c' and a function f and it returns the process p. 'c' should be a positive integer that determines the partitioning of the finite input signal s. The output of the partition determines how many events are consumed by the process during each evaluation cycle. The function f defines the functionality of the process. It takes a partition of the input signal as argument and produces the output events. p is a process that takes one input signal s and generates one output signal ś.

Definition 3.1.1 *mapU (c, f) = p, where p(s) = ś with,*

$$\Psi(v,s) = \langle a_i \rangle \; where \; v(i) = c,$$
$$f(a_i) = \acute{a}_i,$$
$$s, \acute{s}, a_i, \acute{a}_i \in S, i \in N$$

Listing 3.1: Implementation of a map-based process constructor

```
1 (* map-based process constructor for the UMoC *)
2 fun mapU (c,f) = fn (signal) => constructor (c,f,signal)
3
4 fun constructor (_,f,[]) = []
5 | constructor (c,f,s) = f(take(s,c)) @ constructor (c,f,drop(s,c))
```

scan-based process constructor: scan-based process constructor: It instantiates a process with an internal state, which is directly visible at the output as a single event. It takes an initial state, a function that determines the size of each partition based on the current state and a second function that takes a partition and the current state for each activation and generates an output that is the next initial state. Thus, the current state determines how many events are consumed for the next evaluation cycle.

Definition 3.1.2 *scanU* $(\gamma, g, \omega_0) = p$, *where* $p(s) = \acute{s}$ *with,*

$$\Psi(v, s) = \langle a_i \rangle \text{ where } v(i) = \gamma(\omega_i),$$
$$g(a_i, \omega_i) = \omega_{i+1},$$
$$\langle \omega_{i+1} \rangle = \acute{a}_i$$
$$s, \acute{s}, a_i, \acute{a}_i \in S, \omega_i \in E, i \in N$$

Listing 3.2: Implementation of a scan-based process constructor

```
1  (* scan-based process constructor for the UMoC *)
2  fun scanU (h,g,w)= fn (s) => constructor (h,g,w,s)
3
4  fun constructor (_,_,_,[]) = []
5  |   constructor (h,g,w,s)  = [g (w, partition([h w], s))]
6  @   constructor (h,g,g (w,partition([h w],s)),drop (s,(h w)))
```

mealy-based process constructor: It resembles a mealy-based state machine with the addition of a next-state function, an output encoding f that depends on both the input partition and the current state.

Definition 3.1.3 *mealyU* $(\gamma, g, f, \omega_0) = p$, *where* $p(s) = \acute{s}$ *with,*

$$\Psi(v, s) = \langle a_i \rangle \text{ where } v(i) = \gamma(\omega_i),$$

$$g(a_i, \omega_i) = \omega_{i+1},$$
$$f(\omega_i, a_i) = \acute{a}_i,$$
$$s, \acute{s}, a_i, \acute{a}_i \in S,\ \omega_i \in E,\ i \in N$$

Listing 3.3: Implementation of a mealy- based process constructor

```
1  (* mealy-based process constructor for the UMoC *)
2  fun mealyU (h,g,f,w) = fn (s) => constructor (h,g,f,w,s)
3
4  fun constructor (_,_,_,_,[]) = []
5  |   constructor (h,g,f,w,s) = f (w,head (partition([h w],s)))
6  @   constructor (h,g,f,g (w,partition([h w],s)),drop (s,(h w)))
```

moore-based process constructor: It resembles a moore-based state machine with the addition of a next-state function, an output encoding f that depends only on the current state.

Definition 3.1.4 *mooreU* $(\gamma, g, f, \omega_0) = p$, *where* $p(s) = \acute{s}$ *with,*

$$\Psi(v, s) = \langle a_i \rangle \text{ where } v(i) = \gamma(\omega_i),$$
$$g(a_i, \omega_i) = \omega_{i+1},$$
$$f(\omega_i) = \acute{a}_i$$
$$s, \acute{s}, a_i, \acute{a}_i \in S,\ \omega_i \in E,\ i \in N$$

Listing 3.4: Implementation of a moore-based process constructor

```
1  (* moore-based process constructor for the UMoC *)
2  fun mooreU (h,g,f,w) = fn (s) => constructor (h,g,f,w,s)
3
4  fun constructor (_,_,_,_,[]) = []
5  |   constructor (h,g,f,w,s)  = f(w)
6  @   constructor (h,g,f,g (w,partition([h w],s)),drop (s,(h w)))
```

zip-based process constructor: It takes two input signals and generates a signal of 2-tuple events. The first tuple contains the event sequence from the input signal s_a and the second tuple contains the event sequence from the input signal s_b. We can categorize the zip-based constructor on how they perform the zipping into three type of constructors namely zipU, zipUs and zipWithU.

zipU defines a constructor that joins the inputs together based on two functions γ_a and γ_b which take their arguments from a third signal (a control signal). zipUs is a simplified version which has no control input but two constants as parameters that define the partitioning of the input signals. zipWithU is a constructor that allows the joining of the input signals with an arbitrary function. The implementation of the zip-based constructors are shown in Listing 3.5.

Definition 3.1.5 *zip-based processes are defined as:*

$$
\begin{gathered}
zipU\ (\gamma_a, \gamma_b) = p, \textit{ where } p(s_a, s_b, s_c) = \acute{s}, \\
\Psi(v_1, s_a) = \langle a_i \rangle \textit{ where } v_1(i) = \gamma_a(c_i), \\
\Psi(v_2, s_b) = \langle b_i \rangle \textit{ where } v_2(i) = \gamma_b(c_i), \\
\Psi(v_3, s_c) = \langle c_i \rangle \textit{ where } v_3(i) = 1, \\
\langle a_i, b_i \rangle = \acute{e}_i \\
zipUs\ (c_1, c_2) = p, \textit{ where } p(s_a, s_b) = \acute{s}, \\
\Psi(v_1, s_a) = \langle a_i \rangle \textit{ where } v_1(i) = \gamma_a(c_1), \\
\Psi(v_2, s_b) = \langle b_i \rangle \textit{ where } v_2(i) = \gamma_b(c_2), \\
\langle a_i, b_i \rangle = \acute{e}_i \\
zipWithU\ (c_1, c_2, f) = p, \textit{ where } p(s_a, s_b) = \acute{s}, \\
f((a_i, b_i)) = \acute{e}_i, \\
\Psi(v_1, s_a) = \langle a_i \rangle \textit{ where } v_1(i) = \gamma_a(c_1), \\
\Psi(v_2, s_b) = \langle b_i \rangle \textit{ where } v_2(i) = \gamma_b(c_2),
\end{gathered}
$$

$$s_a, s_b, s_c, \acute{s}, a_i, b_i, \in S, \acute{e}_i \in E, i \in N$$

Listing 3.5: Implementation of zip-based process constructors

```
1  (* zip-based process constructor for the UMoC *)
2  fun zipU(f1,f2) = fn (s1,s2,s3)  =>
3  constructor1 (f1,f2,s1,s2,s3)
4
5  fun constructor1 (_,_,[],_,_) = []
6  |   constructor1 (_,_,_,[],_) = []
7  |   constructor1 (_,_,_,_,[]) = []
8  |   constructor1 (f1,f2,h1::t1,h2::t2,h3::t3) = [[f1 (h3,h1::t1)]
9  @ [f2 (h3,h2::t2)]]
10 @   constructor1 (f1, f2,drop (h1::t1,h3),drop(h2::t2,h3),t3)
11
12 fun zipUs(a,b) = fn (s1,s2) => constructor2 (a,b,s1,s2)
13
14 fun constructor2 (_,_,[],_) = []
15 |   constructor2 (_,_,_,[]) = []
16 |   constructor2 (a,b,h1::t1,h2::t2) = [[partition([a],h1::t1)]
17 @ [partition([b],h2::t2)]]
18 @ constructor2 (a,b,drop(h1::t1,a),drop(h2::t2,b))
19
20 fun zipWithU(a,b,f) = fn (s1,s2) => constructor3 (a,b,f,s1,s2)
21
22 fun constructor3 (_,_,_,[],_) = []
23 |   constructor3 (_,_,_,_,[])= []
24 |   constructor3 (a,b,f,h1::t1,h2::t2) = [[f (a,h1::t1)]
25 @ [f (b,h2::t2)]]
26 @   constructor3 (a,b,f,drop(h1::t1,a),drop(h2::t2,b))
```

unzip-based process constructor: It performs the reverse operation of a zip-based process, i.e., it extracts two output signals from a zipped signal based on how it was joined.

Definition 3.1.6 *unzipU ( ) = p, where p(s) = $\langle \acute{s}, \acute{\acute{s}} \rangle$ with,*

$$e_i = \langle \acute{a}_i, \acute{\acute{a}}_i \rangle,$$

$$\Psi(v_1, \acute{s}) = \langle \acute{a}_i \rangle \; where \; v_1(i) = length(\acute{a}_i),$$
$$\Psi(v_2, \acute{\acute{s}}) = \langle \acute{\acute{a}}_i \rangle \; where \; v_2(i) = length(\acute{\acute{a}}_i),$$
$$s, \acute{s}, \acute{\acute{s}} \; a, \acute{a}, \acute{\acute{a}} \in S, e_i \in E, i \in N$$

Listing 3.6: Implementation of unzip process constructors

```
1  (* unzip-based process constructor for the UMoC *)
2  fun unzipU() = fn (s) => constructor(s)
3
4  fun constructor []  = []
5  |   constructor (s) = [first (s), second (s)]
6
7  fun first [] = [] | first s = head (head s) @ first (tail s)
8
9  fun second [] = []
10 |   second s = head (tail (head s)) @ second (tail s)
```

scand-based process constructor: It behaves identically to a scan-based process with the addition that it also emits its initial state.

Definition 3.1.7 *scandU* $(\gamma, g, \omega_0) = p$, *where*

$$p(s) = \langle \omega_0 \rangle \oplus scanU\ (\gamma, g, \omega_0)\ (s),$$
$$s \in S,$$
$$\oplus \; is \; the \; concatenation \; operator$$

Listing 3.7: Implementation of scand-based process constructors

```
1  (* scand-based process constructor for the UMoC *)
2  fun ScandU (f1,f2,w) = fn (signal) => [w] @ (ScanU(f1,f2,w) signal)
```

source-based process constructor: It initializes a signal with events. It takes an initial state and a function encoding which depends on the current state to determine the output event to initialize a signal.

Definition 3.1.8 *sourceU* $(g, \omega_0) = p$, *where* $p(\,) = \acute{s}$ *with,*

$$\omega_i = \acute{e}_i,$$
$$\Psi(v, \acute{s}) = \langle \acute{e}_i \rangle \textit{ where } v(i) = length(g(\omega_{i-1})),$$
$$g(\omega_i) = w_{i+1},$$
$$\acute{s} \in S, \acute{e}_i, \omega_i \in E, i \in N$$

sink-based process constructor: It initializes a signal to an empty signal. It takes an initial state, a function that based on the current state determines how many events need to be consumed for each evaluation cycle and a function encoding that determines the next state.

Definition 3.1.9 *sinkU* $(\gamma, g, \omega_0) = p$, *where* $p(s) = \langle\rangle$ *with,*

$$\Psi(v, s) = \langle a_i \rangle \textit{ where } v(i) = \gamma(\omega_i),$$
$$g(\omega_i) = \omega_{i+1} \textit{ for } s \in S, \omega_i \in E, i \in N$$

init-based process constructor: It initializes a signal with another signal. It takes an event sequence as its argument and appends it to the input signal to create a new signal.

Definition 3.1.10 *initU* $(r) = p$, *where* $p(s) = r \oplus s$ *for* $r, s \in S$

Listing 3.8: Implementation of process constructors to initialize

```
1  (* Process Initiators *)
2  fun sourceU (g,w) = w::SourceU(g,(g w))
3
4  fun sinkU (h,g,w) = fn (s) => constructor1 (h,g,w,s)
5
```

```
6  fun constructor1 (h,g,w,[]) = []
7  |   constructor1 (h,g,w,s)  = constructor1 (h,g,(g w),drop (s,(h w)))
8
9  fun initU (r) = fn (s) => constructor2 (r,s)
10
11 fun constructor2 ([],[])     = []
12 |   constructor2 ([],x::y)   = x::constructor2 ([],y)
13 |   constructor2 (h::t,x::y) = h::constructor2 (t,x::y)
```

3.1.2 Process Combinators

We define compositional operators to combine different processes to form complex processes and process networks. These are also implemented as higher-order functions. The sequential, parallel and feedback operators are shown in Figure 3.1. We discuss the implementation details with respect to finite signals.

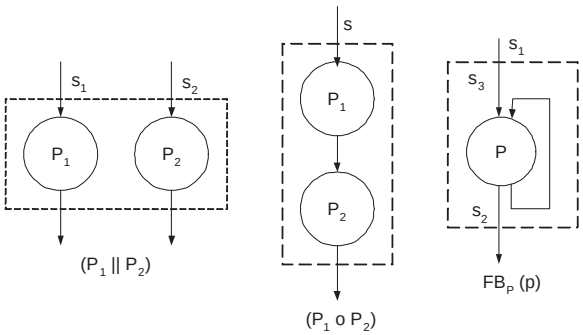


Figure 3.1: Parallel, Sequential and Feedback Operators

parallel composition operator: Let p_1 and p_2 be two processes with one input and one output each, and let s_1,$s_2 \in$ S be two signals. Their parallel composition denoted by $p_1 \parallel p_2$, is defined as follows:

Definition 3.1.11 $(p_1 \parallel p_2)(\langle s_1, s_2 \rangle) = \langle p_1(s_1), p_2(s_2) \rangle$

sequential composition operator: Let p_1 and p_2 be two processes with one input and one output each, and let s ∈ S be a signal. Their sequential composition denoted by $p_1 \circ p_2$, is defined as follows:

Definition 3.1.12 $(p_1 \circ p_2)(s) = p_2(p_1(s))$

Listing 3.9: Parallel and Sequential process combinators

```
1  (* parallel composition operator *)
2  fun parcomp (p1,p2) = fn (s1,s2) => [[p1 (s1)],[p2 (s2)]]
3
4  (* sequential composition operator *)
5  fun seqcomp (p1,p2) = fn (s) => p2 (p1 (s))
```

feedback composition operator: Given a process p: (S×S) → (S×S) with two input signals and two output signals, we define the process FB_p(p) : S → S. The behavior of the process FB_p(p) is defined by the least fixed-point semantics.

Definition 3.1.13 *$FB_p(p)(s_1) = s_2$, where $p(s_1, s_2) = (s_2, s_3)$*

Listing 3.10: Feedback process combinator

```
1  (* Feedback Operator *)
2  fun fb (p) (s) = fixpt(p, s, [], length(s) + 1)
3
4  fun fixpt (q, s, sout, 0) = sout
5  |   fixpt (q, s, sout, n) = fixpt (q,s,(q s sout),n - 1)
```

All the above process constructors and process combinators for the UMoC is also implemented for infinite signals.

3.1.3 Formalized Definition of UMoC

Definition 3.1.14 ***UMoC*** *is defined as MoC = (C,O), where*

$$C = \{mapU, scanU, scandU, mealyU, mooreU, zipU, zipUs, zipWithU, unzipU, sourceU, sinkU, initU\}$$

$$O = \{||, \circ, FB_p\}$$

Listing 3.11: UMoC in the **SML-Sys** Framework

```
1  (* Formulation of the untimed MoC *)
2
3  structure UMoC = struct
4  exception Empty
5  exception NotWellDefined
6  .
7  .
8    (* Definitions of the process constructors *)
9  .
10 .
11   (* Definitions of the process combinators *)
12 .
13 .
14 end
```

We discuss the implementation of an untimed FIR model in Listing A.1 of Appendix A.

3.2 Synchronous Model of Computation (SMoC)

SMoC has two variants that partition the time line into intervals or clock cycles. The clocked synchronous MoC assumes that every evaluation of a process takes one cycle.

Whereas, the perfectly synchronous MoC assumes that no time advances during the evaluation of a process.

3.2.1 Perfectly Synchronous Model of Computation (PSMoC)

We develop perfectly synchronous processes as a special case of untimed processes. Synchronous processes have two specific characteristics. First, they consume and produce exactly one event on each input or output signal during each evaluation cycle. Secondly, E carries the special value $\sqcup$, which denotes the absence of an event. All synchronous process constructors and processes operate exclusively on synchronous signals.

Definition 3.2.1 *Perfectly synchronous process constructors (suffixed with S) are defined as*

$$mapS\ (f) = mapU\ (1, f) = p$$
$$scanS\ (g, \omega_0) = scanU\ (1, g, \omega_0) = p$$
$$scandS\ (g, \omega_0) = scandU\ (1, g, \omega_0) = p$$
$$mealyS\ (g, f, \omega_0) = mealyU\ (1, g, f, \omega_0) = p$$
$$mooreS\ (g, f, \omega_0) = mooreU\ (1, g, f, \omega_0) = p$$
$$zipS() = zipU(1,1) = p$$
$$zipWithS\ (f) = zipWithU\ (1, 1, f) = p$$
$$unzipS\ () = unzipU\ () = p$$
$$sourceS\ (g, \omega_0) = sourceU(g, \omega_0)$$
$$sinkS(g, \omega_0) = sinkU\ (1, g, \omega_0) = p$$
$$initS\ (\bar{r}) = initU(\bar{r}) = p$$

We illustrate the map-based synchronous process constructor.

Definition 3.2.2 *mapS (f) = mapU (1,f) = p, where* $p(\bar{s}) = \bar{s}'$ *with,* $\exists \bar{e} \in \bar{E}$ *then* $f(\sqcup) = \sqcup$ *and* $\forall \bar{e} \in \bar{E}$ *then length(f (ē)) = 1*

Listing 3.12: Implementation of map-based synchronous process constructor

```
1  (* map-based process constructor for the SMoC *)
2  fun mapS (f) = fn (s) => constructor (f,s)
3
4  fun constructor (f,[])    = []
5  |   constructor (h::t,f) = [f h] @ constructor (t,f)
```

The parallel and sequential combinators are similar to the operators defined for the UMoC. However, for the feedback in SMoC, we need to find the fix-point solution for each evaluation cycle separately. So we extend the domain of absent events with another special element, $\perp$ to capture the situation when we do not know if the event occurred or which value it has: $\bar{E}_\perp = \bar{E} \cup \perp$.

Definition 3.2.3 *The order relation* $\preceq$ *on* $\bar{E}_\perp$ *which means "less defined than or equal to" is defined as* $\perp \preceq \bar{e}$ *and* $\bar{e} \preceq \bar{e}$ *for all* $\bar{e} \in \bar{E}_\perp$

feedback composition operator: Let $\bar{S}_\perp$ be the set of signals consisting of elements from $\bar{E}_\perp$, and let $\bar{S}_\perp^\infty \subseteq \bar{S}_\perp$ be the set of infinite signals. For signals of length n, we define $\langle \bar{e}_1, \cdots, \bar{e}_n \rangle \preceq \langle \bar{e}_1', \cdots, \bar{e}_n' \rangle$ iff $\bar{e}_i \preceq \bar{e}_i'$ for all i, $1 \leq i \leq n$

Given a process p: $(\bar{S}_\perp^\infty \times \bar{S}_\perp^\infty) \rightarrow (\bar{S}_\perp^\infty \times \bar{S}_\perp^\infty)$ with two input signals and two output signals, we define the process FB_s (p): $\bar{S}_\perp^\infty \rightarrow \bar{S}_\perp^\infty$ as follows:

Definition 3.2.4 $FB_s\ (p)(\bar{s}_1) = \bar{s}_2$, *where* $p\ (\bar{s}_1, \bar{s}_3) = (\bar{s}_2, \bar{s}_3)$

The events in $\bar{s}_3$ in each evaluation cycle are determined by the least fix-point, that satisfies the constraint of the feedback loop.

Definition 3.2.5 ***PSMoC*** *is defined as MoC = (C,O), where*

$$C = \{mapS, scanS, scandS, mealyS, mooreS, zipS, zipWithS, unzipS, sourceS, sinkS, initS\ \}$$
$$O = \{\ ||, \circ, FB_s\ \}$$

We discuss the implementation of a synchronous SOBEL model in Listing A.2 of Appendix A.

3.2.2 Clocked Synchronous Model of Computation (CSMoC)

For clocked synchronous processes we introduce a delay function $\triangle$, which delays each input by one cycle.

Definition 3.2.6 $\triangle = scandS\ (g, \sqcup)$, *where* $g\ (\omega, \bar{e}) = \bar{e}$

Listing 3.13: Implementation of the delay function

```
1 (* delay process constructor for CSMoC *)
2 fun delta () = fn (s) => scandS(f,0) (s)
```

Using the delay process, we formally define the process constructors (suffixed by *CS*) for CSMoC. The process combinators of CSMoC are similar to the process combinators of PSMoC. We model the SOBEL in this domain that is similar to the implementation of the SOBEL operator in PSMoC as shown in Listing A.2 of Appendix A.

Definition 3.2.7 *Clocked synchronous process constructors (suffixed with CS) are defined as*

$$mapCS\ (f) = mapS\ (f)\ o\ \triangle$$

$$scanCS\ (g, \omega_0) = scanS\ (g, \omega_0)\ o\ \triangle$$
$$mealyCS\ (g, f, \omega_0) = mealyS\ (g, f, \omega_0)\ o\ \triangle$$
$$mooreCS\ (g, f, \omega_0) = mooreS\ (g, f, \omega_0)\ o\ \triangle$$
$$zipCS\ (\ )\ (\ \bar{s_1}, \bar{s_2}\ ) = zipS\ ()\ (\ \triangle(\bar{s_1}), \triangle(\bar{s_2})\ )$$
$$zipWithCS\ (f)\ (\bar{s_1}, \bar{s_2}) = zipWithS\ (f)\ (\ \triangle(\bar{s_1}), \triangle(\bar{s_2})\ )$$
$$unzipCS\ () = unzipS\ ()\ o\ \triangle$$
$$sourceCS = sourceS$$
$$sinkCS = sinkS$$
$$initCS = initS$$

Definition 3.2.8 ***CSMoC*** *is defined as MoC = (C,O), where*

$$C = \{mapCS, scanCS, scandCS, mealyCS, mooreCS, zipCS, zipWithCS, unzipCS, sourceCS,$$
$$sinkCS, initCS\ \}$$
$$O = \{\ ||, \circ, FB_s\ \}$$

3.3 Timed Model of Computation (TMoC)

Previously defined UMoC and SMoC does not follow the notion of physical time. The UMoC is based on data dependencies whereas SMoC has cycle time, in which the computation of the outputs and the next state is "fast enough" and the communication of events do not take any observable time. This is convenient when we are only concerned with the sequence of states and transitions. Nevertheless, there are many cases where we want to take into account the precise timing behavior. The TMoC is used to model the exact timing of all computation and communication. This is the least abstract MoC, and hence allows us to model a process to its minute details in terms of computational

time making it better reflect physical reality. The process constructors for the TMoC, are suffixed with *T*. The timed processes are a blend of untimed and synchronous processes in that they consume and produce more than one event per cycle with the possibility of an event being absent. In addition, they have to comply with the constraint that output events cannot occur before input events for each evaluation cycle. We achieve this in our implementation by enforcing equal number of input and output events for each evaluation cycle. Definition and implementation for some process constructors are shown below.

Definition 3.3.1 *From [8], mealyT is a process constructor that, given γ, f, g, and ω_0 as arguments, instantiates a process p: $\hat{S} \rightarrow \hat{S}$ that is defined as*

$$\begin{gathered} \textit{mealyT } (\gamma, g, f, \omega_0) = p, \textit{ where } p(\hat{s}) = \hat{s}' \textit{ with,} \\ \Psi(v, \hat{s}) = \langle \hat{a}_i \rangle \textit{ where } v(i) = \gamma(\omega_i), \\ g(\hat{a}_{i-1}, \omega_{i-1}) = \omega_i, \\ f(\omega_i, \hat{a}_{i-1}) = \hat{c}_i, \\ \hat{b}_i = \langle \sqcup \rangle^{K_i} \oplus \hat{c}_i \\ \hat{s}, \hat{s}', \hat{a}, \hat{b} \in S, \omega_i \in E, i \in N \end{gathered}$$

The output event for the i^{th} evaluation cycle is given b_i, where $\langle \sqcup \rangle^n$ denotes the sequence of n $\sqcup$ events. The output sequence b_i consist of the result of function f and a number of absent events constituting the delays whenever necessary.

The number of delay events to be inserted for an evaluation cycle is determined by the sequence K_i, defined as

$$K_i = \begin{cases} T_i(i) - T_o(i-1) - 1 & \text{if i} \geq 0 \\ T_i(i) - 1 & \text{if i} = 0 \end{cases}$$

$$T_i(i) = \sum_{j=0}^{i} \text{length}(\hat{a}_j)$$
$$T_o(i) = \sum_{j=0}^{i} \text{length}(\hat{b}_j)$$

Listing 3.14: Implementation of the mealy based process constructor for the Timed MoC

```
1  (* mealy-based process constructor for the TMoC *)
2  fun mealyT (h,g,f,w) = fn (s) => cons2 (h,g,f,w,s,0,0)
3
4  fun cons2 (h,g,f,w,s,len1,len2) = cons1 (h,g,f,w,s,len1+(h w),len2,0)
5
6  fun cons1 (_,_,_,_,[],_,_,_) = []
7  |   cons1 (h,g,f,w,s,len1,len2,iter) =
8  make(getsize(len1,len2,iter)) @ f(w,head (partition([h w], s)))
9  @ cons1 (h,g,f,g(w,head (partition([h w],s))),drop (s,(h w)),
10 len1 + h (g (w,head (partition([h w], s)))),
11 len2 + length (f (w,head (partition([h w],s)))) +
12 length (make (getsize (len1,len2,iter))),iter+1)
13
14 fun getsize (len1,len2,0) = len1 - 1
15 |   getsize (len1,len2,_) = len1 - len2 - 1
16
17 fun make 0 = [] | make n = 0::make (n-1)
```

Definition 3.3.2 *From [8], zipT is a constructor that, given γ as argument, instantiates a process p: $\hat{S} \times \hat{S} \times \hat{S} \rightarrow \hat{S}$, defined as*

$$\begin{gathered}
zipT\ (\gamma) = p, \text{ such that } p(\hat{s}_a, \hat{s}_b, \hat{s}_c) = \breve{s} \text{ where,} \\
\Psi(v_1, \hat{s}_a) = \langle \hat{a}_i \rangle \text{ where } v_1(i) = \gamma(K_i), \\
\Psi(v_2, \hat{s}_b) = \langle \hat{b}_i \rangle \text{ where } v_2(i) = \gamma(K_i), \\
\Psi(v_3, \hat{s}_c) = \langle \hat{c}_i \rangle \text{ where } v_3(i) = \gamma(K_i), \\
\breve{e}_0 = \sqcup, \\
\breve{e}_{i+1} = \langle \hat{a}_i, \hat{b}_i \rangle, \\
\hat{b}_i = \langle \sqcup \rangle^{K_i - 1} \oplus \breve{e}_i,
\end{gathered}$$

$$\textit{where } K_{i+1} = \hat{c}_i[1] \textit{ and } K_0 = 0,$$
$$\hat{s}_a, \hat{s}_b, \hat{s}_c, \tilde{s}, \hat{a}_i, \hat{b}_i\hat{c}_i \in S, \omega_i \in E, i \in N$$

Listing 3.15: Implementation of the zip based process constructor for the Timed MoC

```
1  (* zip-based process constructor for the TMoC *)
2  fun zipT(f) = fn(s1,s2,s3) => "absent_event"
3  @ constructor (f,s1,s2,s3)
4
5  fun constructor (f,[],_,_) = []
6  |   constructor (f,_,[],_) = []
7  |   constructor (f,_,_,[]) = []
8  |   constructor (f,h1::t1,h2::t2,h3::t3) =
9  (if make(f(h3)-1) = []
10 then
11   [[partition ([f(h3)],h1::t1)] @ [partition ([f(h3)],h2::t2)]]
12 else
13   [[make(f(h3)-1)] @ [partition([f(h3)],h1::t1)]
14   @ [partition ([f(h3)],h2::t2)]]
15 )
16 @ constructor (f,drop(h1::t1,h3),drop(h2::t2,h3),t3)
```

Definition 3.3.3 *unzipT is a process constructor that instantiates a process p: $\hat{s} \rightarrow (\hat{s}_a,\hat{s}_b)$.*

Listing 3.16: Implementation of the unzip based process constructor for the Timed MoC

```
1  (* unzip-based process constructor for the TMoC *)
2  fun unzipT() = fn (s) => constructor (s)
3
4  fun constructor(s) = ["absent_event","absent_event]
5  @ [helper1(s),helper2(s)]
6
7  fun helper1 [] = []
8  |   helper1 l = head (if length(head l) = 2
9  then head s else drop(head s,1)) @ helper1(tail s)
10
11 fun helper2 [] = []
12 |   helper2 (s) = head (tail (if length(head s) = 2 then head s
13 else drop (head s,1))) @ helper2 (tail s)
```

Definition 3.3.4 *The sourceT, sinkT and initT process constructors are defined as:*

$$sourceT\ (g, \omega_0) = p$$
$$sinkT\ (g, \omega_0) = p$$
$$initT\ (\hat{r}) = p$$

Definition 3.3.5 ***TMoC*** *is defined as MoC = (C,O), where*

$$C = \{mealyT, zipT, unzipT, sourceT, sinkT, initT\ \}$$
$$O = \{\ ||, \circ, FB_s\ \}$$

We discuss the implementation of a Traffic Light Controller modeled in TMoC in Listing A.3 of Appendix A.

3.4 Interfacing MoCs

It is evident that separate process networks, of the same MoC type may have a different time structure. So when connecting them, the relation between their respective time structure has to be defined. This relation can be constant and simple or it can vary dynamically. We define the interface processes between the MoC domains when the relation between their time structure is a constant and simple. Based on these interface processes we introduce an integrated MoC, consisting of different MoC domains and interface processes. We do not handle the scenario where the time structure relation is dynamic.

3.4.1 Interfacing similar Computational Models

Consider two process networks in two MoC domains, if both are UMoCs then it is not problematic since both the domains have no timing information. Therefore, the two process networks can be directly connected without an interface. However if both are SMoCs, the elementary timing unit may be different. Since there is no reference to an absolute time in the SMoC, the evaluation cycle may be quite different in both the MoCs. If we assume that the evaluation cycle of one MoC could have a duration constant times the other MoC, say **r**, then we can define the following interface constructors.
Let r be a constant positive integer, r ϵ N.

Definition 3.4.1 *intSup emits r events for each input event, and the function f defines the values of the emitted events. intSup (r, f) = mapU (1, f) with, length(f ($\bar{e}$)) = r*

Definition 3.4.2 *intSdown emits one event for r input events. intSdown (r, f) = mapU (r, f) with, length(f($\bar{a}$)) = 1*

Now, if both are TMoCs, it is similar to the SMoC case. We relate the time base in the different domains to each other by the following interface constructors.

Definition 3.4.3 *intTup (r, f) = intSup (1, f) and intTdown (r, f) = intSdown (r, f)*

Listing 3.17: Implementation of the interface constructor for similar MoC

```
1 (* Up-Rating process to interface SMoC to SMoC *)
2 fun intSdown(r,f) = fn (signal) => UMoC.mapU(r,f) (signal)
3
4 (* Down-Rating process to interface SMoC to SMoC *)
5 fun intSup(r,f) = fn (signal) => untimed_constructor (signal,1,f,r)
6
```

```
7  (* Up-Rating process to interface TMoC to TMoC *)
8  fun intTup(r,f) = fn (signal) => intSup(r,f) (signal)
9
10 (* Down-Rating process to interface SMoC to SMoC *)
11 fun intTdown(r,f) = fn (signal) => intSdown(r,f) (signal)
12
13 fun untimed_constructor ([],_,_,_) = []
14 | untimed_constructor (lt,c,f,r) = f(take(lt,c)) r
15 @ untimed_constructor (drop(lt,c),c,f,r)
```

3.4.2 Interfacing different Computational Models

To connect two networks of different MoCs, the interfaces have to bridge domains with different information content concerning time. Therefore, interfaces have to filter out or insert timing information. We define the interface constructors that add timing information with the prefix *insert* and those that remove timing information with the prefix *strip*.

strip-based Interface Constructor: In the strip-based processes, we remove the timing information that they receive on their input signals. For an untimed signal, the $\sqcup$ events should be removed and the other events are passed to the output in the same order they appear at the input.

Definition 3.4.4 *The constructor stripT2U is a process constructor that instantiates a process p:* $\hat{S} \rightarrow \dot{S}$ *which is defined as stripT2U () = p , where p* $(\hat{s}) = \dot{s}$

$$\dot{a}_i = \begin{cases} \langle\rangle & \text{if } \dot{e}_i = \sqcup \\ \langle \dot{e}_i \rangle & \text{otherwise} \end{cases}$$

Definition 3.4.5 *The constructor stripS2U is a process constructor that instantiates a process p:* $\bar{S} \rightarrow \dot{S}$ *which is defined as stripS2U = stripT2U*

Definition 3.4.6 *The constructor stripT2S is a process constructor that instantiates a process p:* $\hat{S} \rightarrow \bar{S}$ *which is defined as stripT2S* $(\lambda) = p$ *, where* $p(\hat{s}) = \dot{s}$

$$\bar{e_i} = \begin{cases} \sqcup & \text{if } strip(\hat{a_i}) = \langle\ \rangle \\ lastt(\hat{a_i}) & \text{otherwise} \end{cases}$$

where lastt $(\hat{s})$ *denotes the last non absent event in signal* $\hat{s}$*.*

Listing 3.18: Implementation of the Strip interface constructor

```
1  (* strip-based process to interface TMoC to UMoC *)
2  fun stripT2U() = fn (s) => u_constructor (s)
3
4  fun u_constructor [] = []
5  |   u_constructor (x::y) =
6  ( if x = 0 then u_constructor(y) else x::u_constructor(y))
7
8  (* strip-based process to interface SMoC to UMoC *)
9  fun stripS2U()  = fn (s) => stripT2U() (s)
10
11 (* strip-based process to interface TMoC to SMoC *)
12 fun stripT2S(p) = fn (s) => s_constructor(p,s)
13
14 fun s_constructor(_,[]) = []
15 |   s_constructor(p,s)  = lastt (take(s,p))
16 @ s_constructor (p,drop(s,p))
17
18 fun lastt []  = []
19 |   lastt(lt) = if remove(lt) = [] then [0] else [last(remove(lt))]
```

insert-based Interface Constructor: In the insert-based processes, we inject λ number of events into the output for a given event in the input.

Definition 3.4.7 *insertU2S is a process constructor that instantiate a process p: $\dot{S} \rightarrow \bar{S}$ which is defined as insertU2S (λ) = p , where p ($\dot{s}$) = $\bar{s}$ and $\bar{a}_i = \langle\ \dot{e}_i\ \rangle \oplus \langle\ \sqcup\ \rangle^{\lambda-1}$*

Definition 3.4.8 *insertU2T is a process constructor that instantiate a process p: $\dot{S} \rightarrow \hat{S}$ which is defined as insertU2T = insertU2S*

Definition 3.4.9 *insertS2T is a process constructor that instantiate a process p: $\bar{S} \rightarrow \hat{S}$ which is defined as insertS2T (λ) = p , where p ($\bar{s}$) = $\dot{s}$, $\hat{a}_i = \langle\ \bar{e}_i\ \rangle \oplus \langle\ \sqcup\ \rangle^{\lambda-1}$*

Listing 3.19: Implementation of the Insert interface constructor

```
1  (* insert-based process to interface UMoC to SMoC *)
2  fun insertU2S (p) = fn (s) => s_constructor(p,s)
3
4  fun s_constructor (p,[])    = []
5  |   s_constructor (p,h::t) = h::makeup(p-1) @ s_constructor(p,t)
6
7  (* insert-based process to interface UMoC to TMoC *)
8  fun insertU2T (p) = fn (s) => insertU2S (p) (s)
9
10 (* insert-based process to interface SMoC to TMoC *)
11 fun insertS2T (p) = fn (s) => t_constructor(p,s)
12
13 fun t_constructor (p,[])    = []
14 |   t_constructor (p,h::t) = h::makeup(p-1) @ t_constructor(p,t)
```

The multiple domains with their interfaces are shown in Figure 3.2.

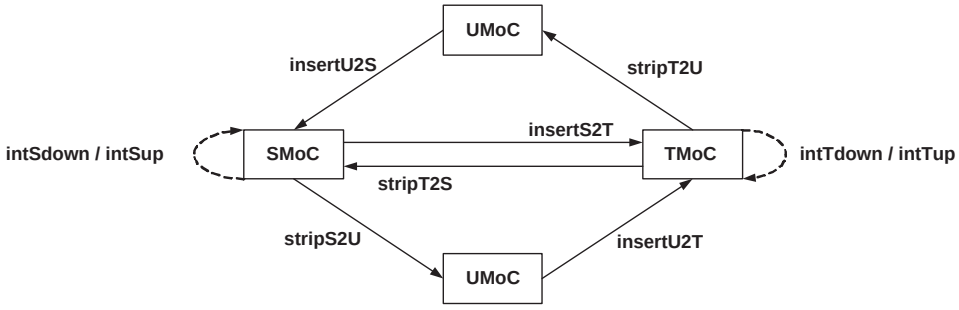


Figure 3.2: Domains with the respective interfaces

We discuss the implementation of a Digital Equalizer System and a Validation Framework in Listing A.4 and A.5 of Appendix A. These systems are built of components modeled in multiple MoCs.

Chapter 4

Refinements in SMoC

For an effective modeling framework to address designs of complex systems, it should start at high levels of abstraction by separating functionality of the system from architecture, communication from computation and promoting the use of formal models and transformations in system design to apply formal verification. This chapter presents design refinement, which is one of the key concepts in our multi-MoC framework. The task of the refinement process is to optimize the abstract specification and to add the necessary implementation details in order to allow for an efficient mapping of the implementation specification onto the target architecture. We achieve the refinements by the application of formally defined transformation rules to the abstract specification that results in a detailed implementation specification. The transformation rules defined are semantic preserving. Each rule has a predefined format and is accompanied with an implication that indicates the change in the semantics caused by a transformation.

4.1 Design Flow

In the **SML-Sys** framework, system modeling starts by first capturing the system behavior into a *Functional Specification* that can be executed using SML. The *functional specification* is then refined into a *Communication Specification* to handle the interface intricacies. Furthermore, it is refined inside the functional domain by stepwise application of well-defined transformation rules into an *Implementation Specification* as shown in Figure 4.1. As the *implementation specification* is a fine-tuned version of the *functional specification*, the same validation and verification methods can be applied to the intermediate versions. Modeling in the **SML-Sys** framework, requires moving large parts of the synthesis, which traditionally are part of the implementation domain, into the functional domain. The task of the refinement process is to optimize the *functional specification* and to add the necessary implementation details in order to allow for an efficient mapping of the *implementation specification* onto the chosen architecture. The *implementation specification* is used to obtain an optimized synthesis of an RTL model [26].

4.1.1 Functional Specification

The specification model is based on the synchronous computational model described in Section 3.2 of Chapter 3. SMoC is built of signals and processes where processes execute concurrently and synchronous communication between them is modeled by signals. A signal is defined as a set of events where each event has a value and a tag i.e., $e \in VxT$. T is a countable set of time stamps and assumed to be the set of natural numbers. Moreover, this MoC is based on the *perfect synchrony hypothesis*, where all signals have the same set of tags. In order to model the absence of a value at a certain tag, we have extended the set of values to hold absent values $\sqcup$. We implement this MoC using process constructors

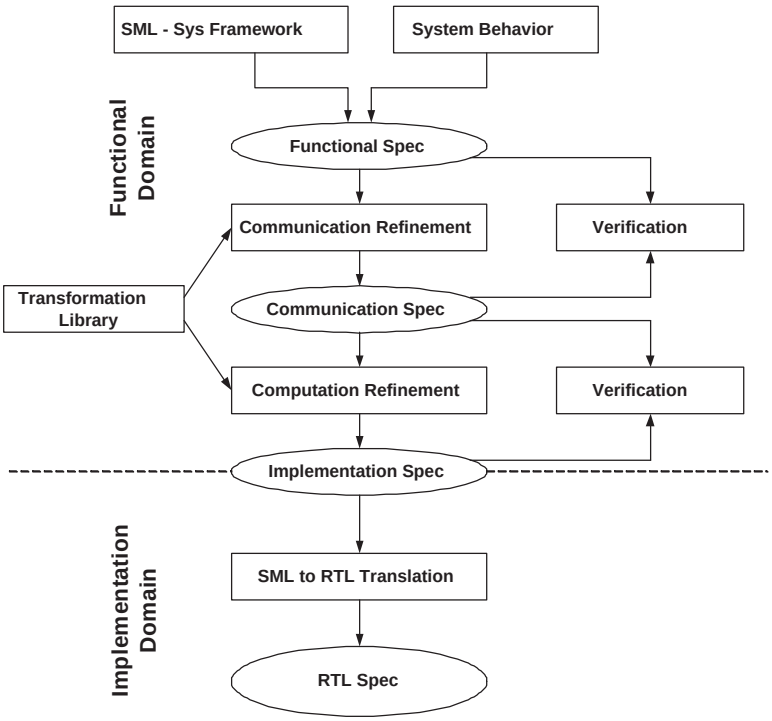


Figure 4.1: Design Flow

and process combinators. In our framework, the modeler must use process constructors for the modeling of processes and process combinators for modeling process networks by composing the different processes. This leads to a well-defined *functional specification*.

Furthermore, we define a set of domain interface constructors that create local synchronous process network with a different signal rate used to establish **synchronous sub-domains**. The domain interface constructors *downDI* and *upDI* generate processes for down- and up-sampling. *downDI(k)* down-samples the input signal with an event rate that is 'k' times slower than the event rate of the input signal 'r' and outputs only each k_{th} input value. *upDI(k)* introduces an output signal with an event rate that is 'k' times higher than the event rate of the input signal 'r' by insertion of 'k - 1' absent events as shown in the Figure 4.2.

The domain interface (DI) constructor *p2sDI(m)* generates a process that transforms parallel input signals into a serial signal with a signal rate that is 'm' times higher than the signal rate of the inputs. The domain interface constructor *s2pDI(n)* generates a process that performs the opposite operation, i.e. the transformation of a serial signal into 'n' parallel signals. The SML code for these DI are shown in Listing B.1 of Appendix B.

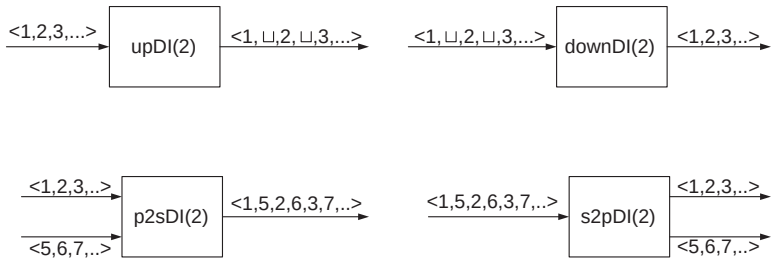


Figure 4.2: Examples of the interface constructors

We define a process groupS_n(k), which reads each input and stores it in an internal state until the state has k values. At that point the state is written to the output and afterwards reset to empty. Since groupS_n is a synchronous process, absent values have to be produced for each input value as long as the grouping is not completed.

Definition 4.1.1 *The process groupS_n (k) is defined as groupS_n(k) = p, where*

$$p(s_1, s_2, \ldots, s_m) = \acute{s} \text{ and } p = moore S_n\ (f, g, \omega_0) \text{ s.t.}$$

$$f((x, y, \ldots, z), \omega_0) = \begin{cases} (x, y, \ldots, z) & \text{if } |\omega_0| = k \vee |\omega_0| = 0 \\ \omega_0 \oplus (x, y, \ldots, z) & \text{otherwise} \end{cases}$$

$$g(\omega_0) = \begin{cases} [\omega_0] & \text{if } |\omega_0| = k \\ \sqcup & \text{otherwise} \end{cases}$$

$$\omega_0 = \langle\ \rangle$$

$\oplus$ *is the concatenation operator.*

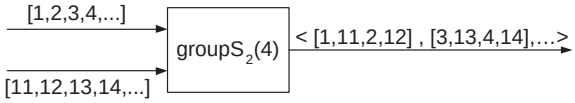


Figure 4.3: Example of the groupS_n constructor

4.1.2 Communication Specification

The *functional specification* uses the same synchronous communication mechanism between all its subsystems. Nevertheless, large systems are usually not implemented as one single unit, but are partitioned into hardware and software blocks communicating with each other via a dedicated communication protocol. Communication refinement is used to partition the system into different hardware/software parts by inserting the appropriate communication interfaces between them. For an architecture geared towards globally asynchronous and locally synchronous (GALS) design, the functional specification with synchronous communication channels between the sub-modules is transformed into an asynchronous protocol which can be modeled as a buffered mode of communication with a handshake. If the rates of the processes communicating are known, then the interface can be refined as a buffered mode of communication without a handshake protocol. We define the *HandshakeWithFIFO* transformation, which does a refinement by introducing a FIFO and a handshake between the two communicating processes. This transformation does a step-wise fine-tuning of a functional specification to a communication specification which preserves semantics. The *communication specification* differs from the *functional specification* in terms of the additional delay that the events incur due to the communication protocol.

HandshakeWithFIFO Transformation

The *HandshakeWithFIFO* transformation procedure is shown in Figure 4.4. This transfor-

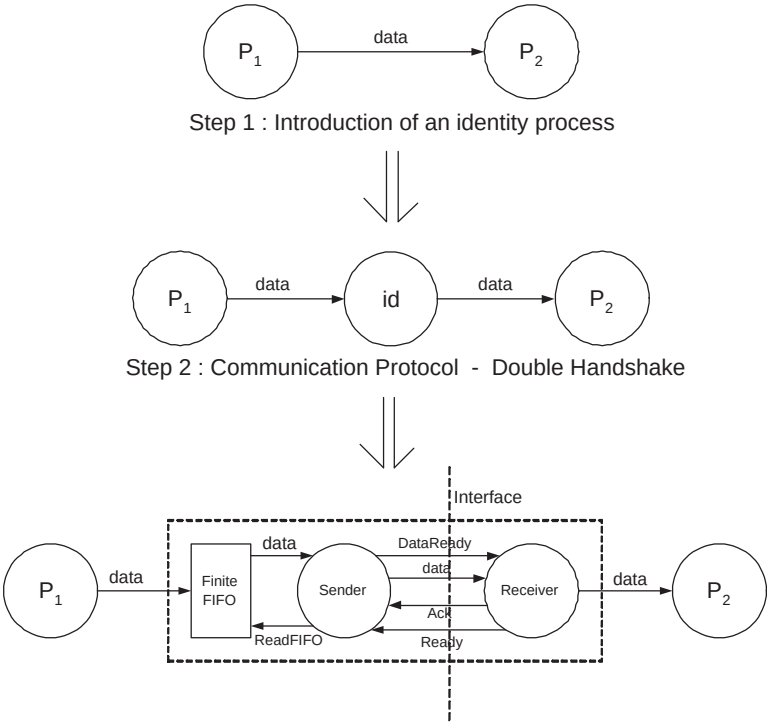


Figure 4.4: *HandshakeWithFIFO* Rule

mation is done in two steps. In the first step an identity process is introduced between the given two processes P_1 and P_2. We introduce an identity process so that while refining the communication the computational processes P_1 and P_2 are not altered. In the second step, the identity process is refined into a handshake protocol, implemented by three processes: FIFO buffer, Sender and Receiver. The process Sender is a Moore state machine and the Receiver is a Mealy state machine. The double handshake protocol works as follows. The Sender initiates a *ReadFIFO* event and tries to read data from the FIFO. If data arrives, then Sender starts a data transmission over the interface, and emits *DataReady* to Receiver. After that, it waits for the reply message *Ready* from Receiver,

which means that Receiver is ready to take the data. When *Ready* appears, Sender transmits the data and waits for an acknowledgment *Ack* from the Receiver signalling that the data was transferred. The handshake protocol implies a delay of one or more cycles for each event, as Sender and Receiver are synchronous processes. This means, that P_2 will handle a delayed sequence of events when compared to the *functional specification*. The implementation of this transformation is shown in Listing B.2 of Appendix B.

4.1.3 Implementation Specification

The *communication specification* now further undergoes a computational refinement where it is transformed into an *implementation specification*. The computational refinement is also done by the application of transformation rules from the library which transform the abstract model into an efficient implementation. These rules are semantic preserving and we finally obtain an optimized *implementation specification*.

4.2 Transformations Library

Definition 4.2.1 *(**Transformation Rule**) [27]*
A transformation rule $\boldsymbol{R : P \rightarrow P}$ *is a mapping of a process network* $\boldsymbol{PN}$ *onto another process network* $\boldsymbol{PN'}$ *with the same number of input and output signals. A transformation rule is denoted by* $\boldsymbol{R(PN) = PN'}$.

Definition 4.2.2 *(**Transformation**) [27]*
A transformation $\boldsymbol{T: (S_0, R) \rightarrow S_1}$ *is a mapping of a system specification* $\boldsymbol{S_0}$ *onto another system specification* $\boldsymbol{S_1}$ *with the same number of input and output signals. Using the transformation rule* $\boldsymbol{R}$*, the internal process network* $\boldsymbol{PN}$ *in* $\boldsymbol{S_0}$ *is replaced by* $\boldsymbol{R(PN)}$ *to yield* $\boldsymbol{S_1}$.

A *transformation* is the application of a transformation rule to a process network that is a part of a system specification as illustrated in Figure 4.5. S_0 is stepwise fine-tuned by well-defined design transformations T_i into a final specification S_n. In Figure 4.5, the

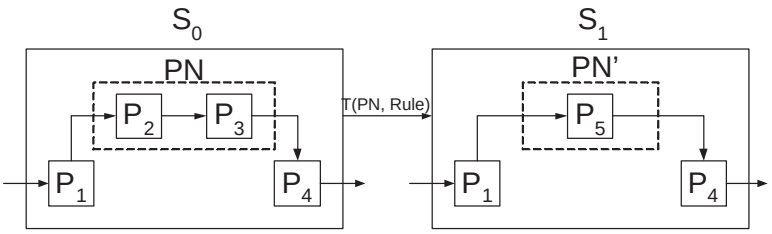


Figure 4.5: Design Transformation

transformation rule is applied to the process network inside the system specification S_0. The result of the transformation is the system model S_1. The only difference between S_0 and S_1 is the replacement of PN by PN′. To compare the transformed specification to the original specification, we introduce a characteristic function which characterizes the functional behavior of a process network.

4.2.1 Design Transformations

The designer applies transformations to a functional specification by choosing transformation rules from the transformation library. The transformation rules are characterized by a name, the required format and constraints of the original process network, the format of the transformed process network and the implication for the design, i.e. the relation between the original and transformed process network.

Definition 4.2.3 ***(Characteristic Function)*** *[27]*

In the synchronous domain, the characteristic function $\Gamma_{PN}\ ((s_1)^r, \ldots, (s_m)^r, i)$ *of a process*

network PN with the input signals $(s_1)^r, \ldots, (s_m)^r$ *and the output signals* $(s'_1)^u, \ldots, (s'_n)^u$ *expresses the dependence of the output events at* tag_i^u *on the input signals.*

$$\Gamma_{PN}(s_1^r, \ldots, s_m^r, i) = ((e'_1)_i^u, \ldots, (e'_n)_i^u)$$

The characteristic function can be derived for any process network including domain interfaces and for each process constructed by a synchronous process constructor or domain interface constructor and for each combinator process. We illustrate the formulation of the characteristic function for a map-based process in Listing 4.1. We have formulated the characteristic functions for all the process constructors and process composition operators except the feedback operator.

Listing 4.1: Characteristic function of a map-based process constructor

```
1 (* Characteristic Function *)
2 fun chfn_mapS (f,j) = fn (s) => fn_map (f,j,s)
3
4 fun fn_map (_,_,[])    = raise No_Characteristic_function_exist
5 |   fn_map (f,1,h::t) = f h
6 |   fn_map (f,j,h::t) = fn_map (f,j-1,t)
```

Transformation Rule 1 - ConstructorMerge

This rule illustrates the merging of the map-based constructor as shown in Listing B.3 of Appendix B. Most of the other constructors can be merged in a similar fashion.

$$\text{mapS (g} \circ \text{f)} = \text{mapS (g)} \circ \text{mapS (f)}$$

The transformation rule *MapMerge* [27] takes a process network of the form

$$PN(\bar{s}) = (\text{mapS (g)} \circ \text{mapS (f)})(\bar{s})$$

and transforms it into the form

$$PN'(\bar{s}) = (\text{mapS (g} \circ \text{f))}(\bar{s})$$

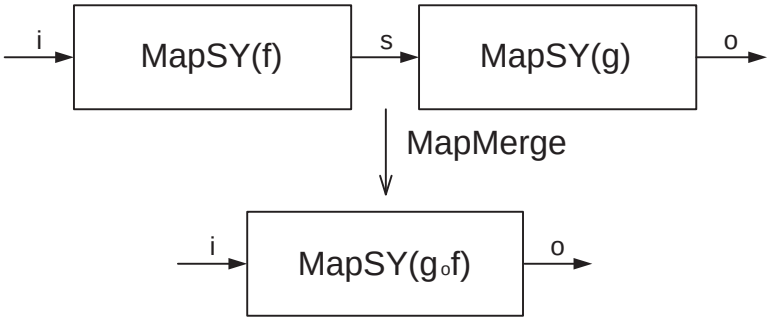


Figure 4.6: Illustration of the transformation rule *MapMerge*

Transformation Rule 2 - BalancedTree

The transformation *BalancedTree* [27] transforms a combinational m-input process into a balanced network of m - 1 processes with two inputs. If the process has a regular structure such as an N-bit adder or multiplier, where N=4,8,16, $\cdots$ the process can be transformed into a balanced network of N-1 2-input processes. This transformation takes an original network of the form

$$\bar{s}' = \text{PN}(\bar{s}_1, \cdots, \bar{s}_m)$$
$$\text{PN}(\bar{s}_1, \cdots, \bar{s}_m) = \Phi_m(\text{f})(\bar{s}_1, \cdots, \bar{s}_m)$$
$$\text{m} = 2^k, \text{k } \varepsilon \text{ N}$$
$$\text{f}(x_1, \ldots, x_m) = x_1 \otimes \cdots \otimes x_m \text{ is associative}$$
$$\Phi_m \text{ is the zipWithS}_m \text{ process constructor}$$

into the form

$$\bar{s}' = PN'(\bar{s}_1, \cdots, \bar{s}_m)$$
$$PN'(\bar{s}_1, \cdots, \bar{s}_m) = \Phi_2(\mathrm{g})(\Phi_2(\mathrm{g}) \ldots (\Phi_2(\mathrm{g})(\ \Phi_2(\mathrm{g})(\bar{s}_1, \bar{s}_2)\ \Phi_2(\mathrm{g})(\bar{s}_3, \bar{s}_4))), \Phi_2(\mathrm{g}) \ldots (\ \Phi_2(\mathrm{g})(\Phi_2(\mathrm{g})(\bar{s}_{m-3}, \ldots, \bar{s}_{m-2}), \Phi_2(\mathrm{g})(\bar{s}_{m-1}, \bar{s}_m))))$$
$$\mathrm{g(x,y)} = \mathrm{x} \oplus \mathrm{y}$$

This transformation takes a structured PN and transforms it to a balanced structure with the reduction in the number of process by 1 from the original PN. It can only be applied for all processes that comply to the format and constraints given in original process network, where the operator $\otimes$ is associative. The implementation is shown in Listing B.4 of Appendix B.

Transformation Rule 3 - PipelinedTree

The *PipelinedTree* [27] transformation pipelines a balanced tree structure of possibly different 2-input processes into a pipelined tree structure as shown in Listing B.5 of Appendix B. It converts a network of the form

$$\bar{s}' = \mathrm{PN}(\bar{s}_1, \ldots, \bar{s}_m)$$
$$\mathrm{PN}(\bar{s}_1, \ldots, \bar{s}_m) = \Phi_2(f_{m-1})(\ \ldots (\Phi_2(f_1)(\bar{s}_1, \bar{s}_2), \ldots\ ), \ldots (\Phi_2(\ \ldots \Phi_2(f_{m/2})(\bar{s}_{m-1}, \bar{s}_m)))$$
$$\mathrm{m} = 2^k, \mathrm{k} > 1$$
$$\Phi_m \text{ is the zipWithS}_m \text{ process constructor}$$

into the form

$$PN'(\bar{s}_1, \ldots, \bar{s}_m) = \Delta_1(\omega_0) \circ \Phi_2(f_{m-1})(\ \ldots (\Delta_1(\omega_0) \circ \Phi_2(f_1)(\bar{s}_1, \bar{s}_2\ ), \ldots), \ldots (\ \ldots, \Delta_1(\omega_0) \circ \Phi_2(f_{m/2})(\bar{s}_{m-1}, \bar{s}_m)))$$

The *PipelinedTree* transformation introduces a delay of k cycles.

Transformation Rule 4 - TwoClockDomain

The *TwoClockDomain* transforms a combinational process with a regular structure into a structure with two separate clock domains. The transformed process network uses an FSM process to schedule the operations into several clock cycles. This transformation is illustrated in Figure 4.7

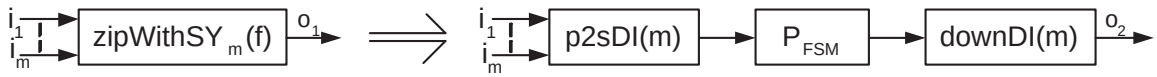


Figure 4.7: Illustration of the transformation rule *TwoClockDomain*

The transformation rule *TwoClockDomain* takes a process network of the form

$$\bar{s}' = \mathrm{P}(\bar{s}_1, \bar{s}_2, \ldots, \bar{s}_m)$$
$$\mathrm{P}(\bar{s}_1, \bar{s}_2, \ldots, \bar{s}_m) = \mathrm{zipWithS}_m(\mathrm{f})(\bar{s}_1, \bar{s}_2, \ldots, \bar{s}_m)$$
$$\mathrm{f}(x_1, x_2, x_3, \ldots, x_m) = g_{m-1}(h_{m-1}(x_m),(\ldots,(g_0(h_1(x_2), h_0(x_1))), \ldots,))$$

and transforms it into the form

$$\bar{s}' = P'(\bar{s}_1, \bar{s}_2, \ldots, \bar{s}_m)$$
$$P'(\bar{s}_1, \bar{s}_2, \ldots, \bar{s}_m) = (\mathrm{downDI(m)} \circ P_{FSM} \circ p2sDI(m))\ (\bar{s}_1, \bar{s}_2, \ldots, \bar{s}_m)$$
$$P_{FSM} = \mathrm{mooreS}\ (\mathrm{u}, \mathrm{w}, \omega_0)$$

$$u(x,(k,s)) = \begin{cases} (1, h_k(x)) & \text{if k = 0} \\ (k+1, g_k(h_k(x), s)) & \text{if 0 < k < m - 1} \\ (0, g_k(h_k(x), s)) & \text{if k = m - 1} \end{cases}$$

$$w(k,s) = \begin{cases} s & \text{k} = 0 \\ \sqcup & 0 < \text{k} < \text{m} \end{cases}$$

$$\omega_0 = (0, \text{initValue})$$

The transformation rule requires the combinational function f to have the format

$$\text{f}(x_1, x_2, x_3, \ldots, x_m) = g_{m-1}(h_{m-1}(x_m),(\ldots,(g_0(h_1(x_2), h_0(x_1))), \ldots,))$$

in order to be able to schedule the computation of f into m steps. In the first step

$$y_0 = h_0(x_1)$$

is calculated and in the following m - 1 steps the intermediate value y_i - 1 is used to calculate the final result y_m-1 according to

$$y_i = g_i(h_i(x_{(i+1)}), y_{i-1})$$

The transformed process network works as follows. During an input event cycle the domain interface p2sDI (m) reads all input values at signal rate 'r' and outputs them as a sequence $\langle x_1, \ldots x_m \rangle$ with the signal rate 'mr' one by one in the corresponding m event cycles. The process P_{FSM} is based on the process constructor *mooreS* and executes the combinational function f of the original process within m cycles. In state 0 the function h_0 is applied to the first input value x_1 and the result is stored as intermediate value s. In the n - 1 following states the function g_i is applied to $h_i(x_i$+1) and the intermediate value. At tag 0, m, 2m, ... the process outputs the intermediate value, otherwise the

output is absent ⊔. The domain interface *downDI(m)* down-samples the input signal to signal rate r and outputs only each m^{th} input value starting with tag 0, thus suppressing the absent values from the output of P_{FSM}. The *TwoClockDomain* transformation delays the output of the transformed process network one event cycle compared to the original process network. The first event of the output signal has the second component of the initial state, the value *initValue*.

4.3 Case Study: SOBEL Operator

The Sobel operator performs a 2-D spatial gradient measurement on an image that is a simple edge detection algorithm. Given a pair of 3 x 3 convolution masks (G_x, G_y) designed to respond maximally to edges (running vertically and horizontally relative to the pixel grid), the operator finds the absolute gradient magnitude at each point in the input gray-scale of an image.

Functional Specification of the SOBEL Operator

The functional specification, shown in Figure 4.8 has three main modules the SOURCE, COMPUTATION and the SINK.

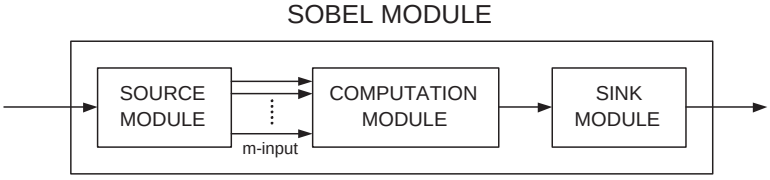


Figure 4.8: Functional Specification of the SOBEL Module

SOURCE Module: The SOURCE outputs the m-row, n-column input as 'l' number of 3

x 3 image matrix to the COMPUTATION module, where m, n >= 3 and l = (m - 2) * (n - 2).

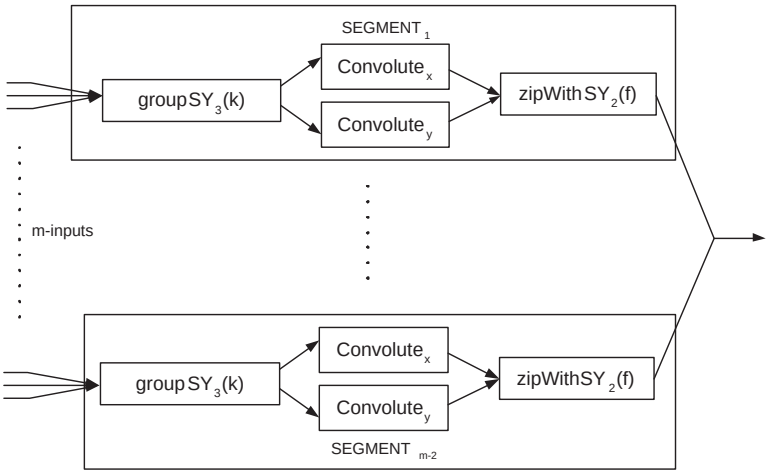


Figure 4.9: The COMPUTATION Module

COMPUTATION Module: This module computes the corresponding intensities of an m x n image. The m-rows are partitioned into groups of three, such that adjacent groups have two common rows and these partitions are given as input to 'm - 2' *groupS$_3$(k)* process constructors. The three inputs are read and stored in the internal state of a *groupS$_3$(k)* process constructor until the state has 'k' values. At that point the state is written to the output and afterwards reset to an empty. Since *groupS$_n$* is a synchronous process, absent values are produced for each input value as long as the grouping is not complete. The output, a vector of size 'k' is given to a process which convolutes it with the horizontal and vertical mask to generate the corresponding gradients G_x and G_y. These are then zipped using a *zipWithS$_2$*(f) process constructor with a function f that sums the squared intensities. This complete flow is called a *segment* as shown in Figure 4.9.

SINK Module: This module is modeled as a sinkS process constructor which is applied to the output of the COMPUTATION module.

Communication Specification of the Sobel Operator

One possible implementation of this system may have a configuration, where system blocks are partitioned into software and hardware. We assume the case where the COMPUTATION unit is implemented in hardware and SINK in software such as a image visualization software. This partit ioning demands an asynchronous protocol between these system blocks instead of the synchronous channel in the *functional specification*. We replace the synchronous channel with a double handshake protocol by refining the interface as shown in Figure 4.10. A prerequisite for this refinement is that the data type of the channel must be equipped to handle absent events (⊔). We apply the rule

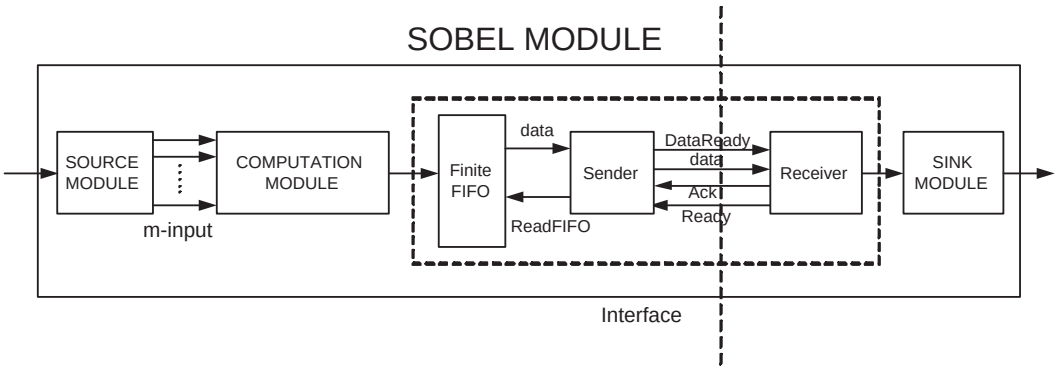


Figure 4.10: Communication Refinement of the Sobel Module

HandshakeWithFIFO which is a transformation done in two steps, in the first step an identity process is introduced between COMPUTATION and SINK, and in the second step, this process is refined into a handshake protocol, implemented by three processes: FIFO buffer, Sender and Receiver. The SINK unit will process a delayed sequence of events when compared to the *functional specification*.

Implementation Specification of the SOBEL Operator

Figure 4.11, shows a segment of the COMPUTATION unit which includes a convolution algorithm. The *ConvoluteWith* processes take a vector of 9 samples each and produce the corresponding result in the form of collection of size 9. The process *GroupSamples* reads 3 samples from each of the inputs and groups them into a collection of size 9. The computation of this vector takes 3 cycles and serves as inputs for the *ConvoluteWith* processes. Since a synchronous computational model is used in the specification model, the grouping process has to produce an output event for each input event. This results in the output of 2 absent events for each computed k-sized vector. The processes *Con-*

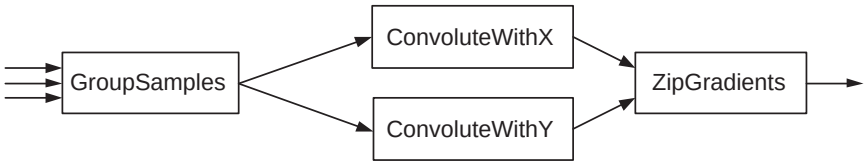


Figure 4.11: A segment of the COMPUTATION

voluteWithX and *ConvoluteWithY* are equipped to handle absent events. This is not a drawback for the specification phase, but a direct feature of the implementation that the convolution can only be done at the 3-rd clock cycle. Instead, the processes have to produce the result during a single cycle and will be idle during 2 clock cycles. This scenario prevails for all the segments of the module. This implementation of the COMPUTATION unit is very inefficient, since the convolution is clearly the most time consuming and will determine the clock period for the whole system and thus the overall system performance.

For a more efficient specification, we refine the *communication specification* into an *implementation specification* by introducing synchronous sub-domains into the system model. The refinement introduces a new clock domain with the help of the domain interface

constructors after the process *G(k)*, in order to filter the absent events and allow for a more efficient use of the C_x *and* C_y processes. The *downDI(3)* process after the *G(k)* as

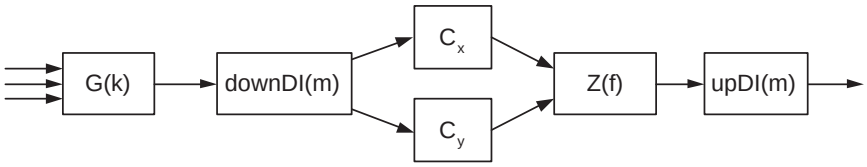


Figure 4.12: Transformation of a segment

shown in Figure 4.12 propagates only every 3-rd event to the output. All absent events are consumed. In order to avoid signal inconsistency which may occur due to this refinement, we introduce another domain constructor *upDI(3)* to insert the absent events at the end of the computation. The implementation of the refined Sobel module is shown in Listing B.7 of Appendix B.

Chapter 5

EWD

To model complex embedded systems, one needs a modeling framework that is easily **customizable** by users, as metamodeling frameworks and is **multi-targeted** in the sense that several special purpose models for synthesis, analysis or verification, can automatically be generated. We therefore, present **EWD**[1], a metamodeling driven customizable multi-MoC system modeling environment, built using GME. It facilitates in customizing MoC-specific visual modeling syntax by providing design-time automatic syntactic and semantic checks on the models for conformance to their metamodels. It provides parsing and translation tools to save such models in an XML-based platform independent language that we call *Interoperable Modeling Language* (IML). These tools convert the IML into SML or Haskell models that are executed and analyzed either by our existing model analysis tools **SML-Sys**, or the ForSyDe [27] environment. Furthermore, it also consists of a tool that generates SMV-based verification models. In this chapter, we illustrate our visual modeling environment and the advantages of having such a multi-MoC model-

[1]Software systems developed by us are codenamed after famous computer scientists. EWD (*e-wood*) stands for E. W. Dijkstra.

ing environment. Moreover, we provide details of the *model construction* and the *code generation* phase of the design flow. Finally, we illustrate the usage of EWD by modeling an adaptive amplifier using the UMoC metamodel.

EWD provides a user with the following features:

1. A visual modeling environment that supports multi-MoC modeling for embedded software and hardware systems.
2. Enforces constraints during models construction by restricting the user to the underlining MoC. Violation of these constraints are checked dynamically.
3. Automated generation of executables for the **SML-Sys** and ForSyDe [38] frameworks.
4. Automated translation of the model to SMV-based partial verification templates.
5. Multi-targeting through an XML-based interoperability that defines a platform independent language and its binding to various target languages.

5.1 The EWD Design Flow

The first step in the **EWD** design flow is the *model construction phase* (MC) followed by the *parsing phase* which populates our *xmlTree* data structure that is a platform-independent representation. Finally, in the *code generation phase* print streams are written that translate the *xmlTree* to provide multi-targeted simulation and verification capabilities.

MC phase: With the given metamodels, the user instantiates a modeling domain which creates the environment for the modeling activity. The user constructs models that are

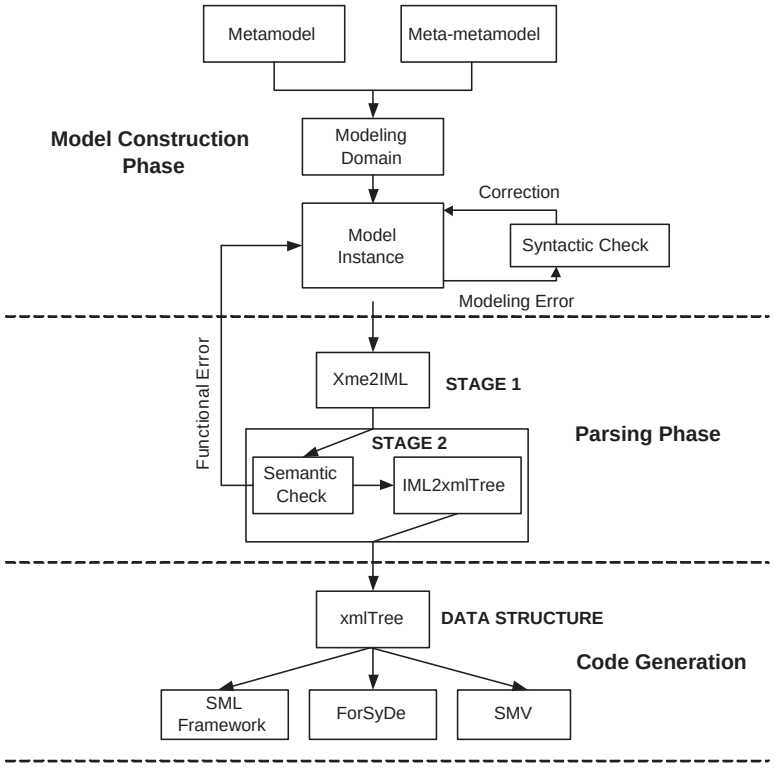


Figure 5.1: The Design Flow

constrained by the semantics of the underlining MoC-specific metamodel. Modeling errors flagged during the syntactic check must be corrected before proceeding to the next phase. Upon completion of the model construction, it is converted to an XML representation and saved in a *.xme* extension file. Failure to correct flagged violations results in an incomplete model that cannot be saved to the XML representation.

Parsing Phase: We extract from the .xme file the model and metamodel relevant information and represent these using a platform-independent language called IML. This extensible modeling syntax provides implementation independence. It is given as input to the second stage where we first perform a check for functional errors in terms of type mismatch and compositional errors. For instance, if two untimed MoC processes with constant but different communication data rates are connected, it can be identified at

this stage, provided the data rates are part of the process types. Upon detecting an error, we retrace back to the model construction phase. Successful completion of the semantic check would result in parsing the IML syntax to populate the *xmlTree* data structure.

Code Generation: The populated *xmlTree* structure is used to generate SML or Haskell-based executable models depending on the target framework. Furthermore, for verification purposes the model can be partially translated to SMV. Thereby, providing the modeler with a multi-targeted modeling environment where one starts with a framework-independent design, which can be simulated and further processed in different design flows. We envision that new design flows can be added with ease in **EWD**. Furthermore, EWD can facilitate a reverse design-flow, where models created in **SML-Sys** or ForSyDe are converted to platform independent IML-based representations.

5.2 Model Construction Phase

Our metamodels describing the generic MoCs are based on the semantic framework in [8]. We have built the metamodel for the Untimed, Synchronous, and Timed MoCs and integrated them into a multi-MoC paradigm as shown in Figure 5.2. Addition of a new metamodel is relatively easy for the following reasons:

1. IML and their corresponding tools do not change.
2. Underlining framework does not change and our metamodel skeleton can be reused.

The design of the metamodel for UMoC shown in Figure 5.3 is made of the following components: processes, process networks (PNs) of type *model* and their functional com-

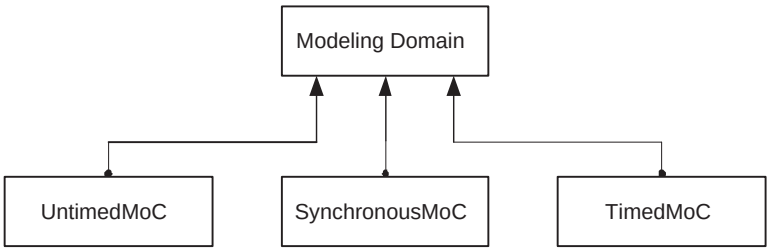


Figure 5.2: The toplevel of the Metamodels in GME

positions defined by a set of process combinators defined within the process network. The design also contains signals of type *atom* which are used to define the inputs and outputs for the instantiated model.

The functionality of different process models is shown in Chapter 3. The metamodel for the SMoC and TMoC is built in a similar manner. Once a metamodel is created, it is registered for conformance to the meta-metamodel of GME as shown in Figure 2.1. The user instantiates a metamodel that sets-up a modeling environment for the user to proceed with the model construction followed by multi-targeted simulation and verification. Composition of metamodels within another metamodel allows us to build meta-theory for hierarchies of MoCs. SLDLs based on informal semantics such as SystemC, do not have a clean semantic theory for multi-MoC hierarchy and compositions. EWD alleviates this problem for any language that is formalized with meta-theory in EWD/GME.

5.2.1 Process Network Structure

A MoC-specific metamodel can be described as a set of process constructors and combinators that are used to create PNs.

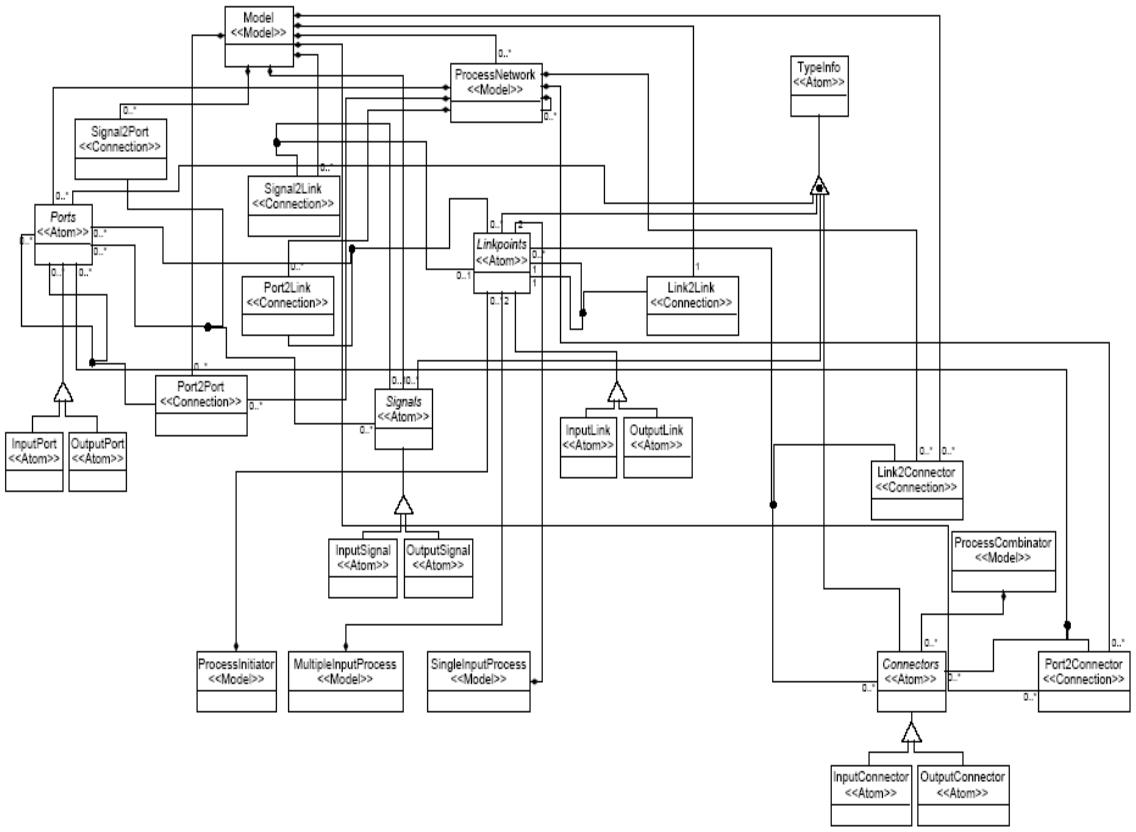


Figure 5.3: An Abstract View of UMoC metamodel

A PN as shown in Figure 5.4 acts as a container for process constructors that interface through a set of process combinators. It also has self containership relation. Designing the PN in this fashion enables it to act as a collection of processes and PNs where an elementary PN is built from two processes and a combinator. PNs also contain *ports* of type *atom*. These ports formulate 3 types of interfaces defined through connection streams namely *Port2Port*, *Port2Signal* and *Port2Connector*. The *Port2Port* stream is used to bridge two PNs whereas the *Port2Connector* stream is used to connect a PN to other processes through a process combination operator to build a bigger PN. The *Port2Signal* stream defines the inputs and outputs to the PN. Defining the following connection

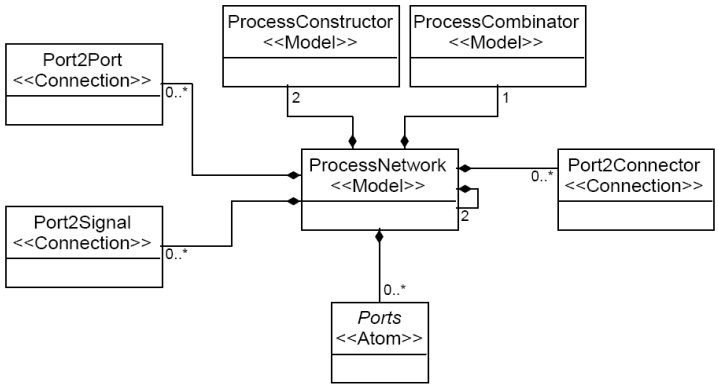


Figure 5.4: PN structure in GME

streams in a PN restricts the modeler from connecting objects which do not conform with these streams. For example, if the modeler wants to compose two processes in a PN without a combinator, the design violates the rules of the metamodel flagging a design-time error.

Table 5.1: Process Constructors

Type	**Constructor Name**	**Functionality of the Constructor**
Single Input Processes (SIP)	map, scan, mealy, moore, scand	Takes one input signal and generates one output signal
Multiple Input Processes (MIP)	zip, unZip, zipWith	Takes 'm' input signals and generates 'n' output signal
Process Initiators (PI)	source, sink, init	Initialize a signal
Type	**Combinator Name**	**Functionality of the Combinator**
Process Combinator (PC)	seqcomp, parcomp, fb	Composes processes/PNs sequentially, parallelly or with feedback

5.2.2 Process Constructor Structure

We define a process constructor of type model to be abstract and is derived as either a Single Input Process (SIP), Multiple Input Process (MIP) or a Process Initiator (PI) constructor as shown in Figure 5.5. An SIP is defined as type *model* and inherited by

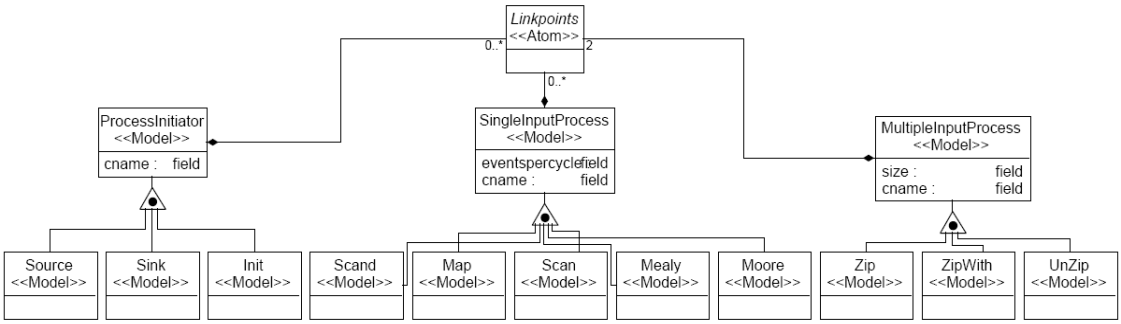


Figure 5.5: Process constructors in GME

the following process constructor - *map, scan, scand, mealy and moore* implying that a SIP can only be instantiated as one of these constructors. It has two attributes namely *cname* and *eventsperCycle*. The attribute *cname* names the constructor that is extracted during the parsing phase and used in translation to map it to the correct target process in the respective framework. The attribute *eventsperCycle* defines the size of the evaluation cycle for the respective process being derived. Similarly an MIP is inherited by the *zip, zipWith and the unzip* processes. It also has two attributes namely *cname* and *size*. *cname* serves the same purpose whereas *Size* indicates the number of inputs for these processes, since the processes can have multiple inputs. Finally, PI is inherited by the respective process initiators *source, sink and init* and has only the *Cname* attribute. The functionalities of these constructors as shown in Table 5.1 are defined in Chapter 3. In Figure 5.6, It can be seen that a process constructor acts as a container for *links*. Links define the inputs

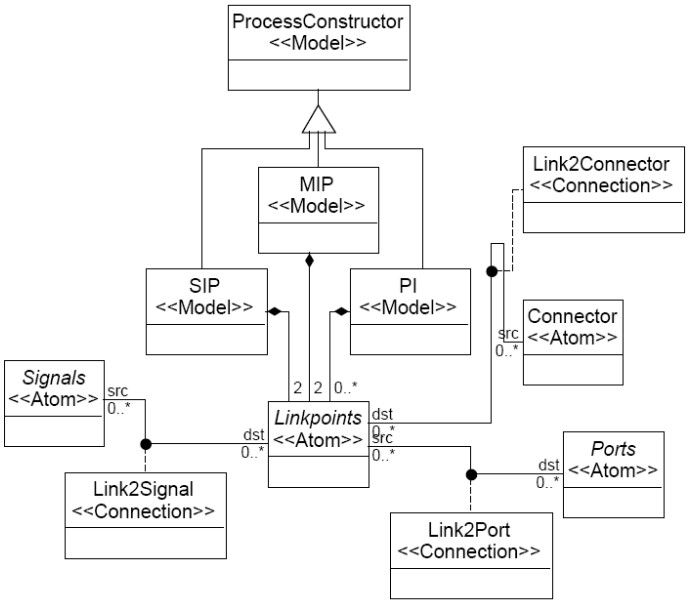


Figure 5.6: Process structure in GME

and outputs of a process with its attributes specifying the name, type, and structure of these links. The process constructor also has a set of interfaces defined to describe the interaction with other elements in the metamodel. These interfaces restrict the user from composing incompatible elements by flagging a violation. The interfaces are modeled as connection streams with a single source/destination and the multiplicity defining the parity of the interaction. The interface streams defined for process constructors are as follows:

1. *Link2Signal* connection
2. *Link2Port* connection
3. *Link2Connector* connection

The *Link2Signal* stream defines the interface between input/output signals and the process. This interface includes the additional constraint on type consistency. Each signal has a type signature and the link of a process has a specific signature of its own. If these signatures do not match, then the composition is not valid and a type inconsistency violation is flagged. The *Link2Port* stream interface is used to construct complex processes or process networks, wherein a process is composed to a PN. It is just another name for the *Port2Link* stream in Figure 5.4. The *Link2Connector* connection defines the interface for composing two processes through a process combinator.

5.2.3 Process Combinator Structure

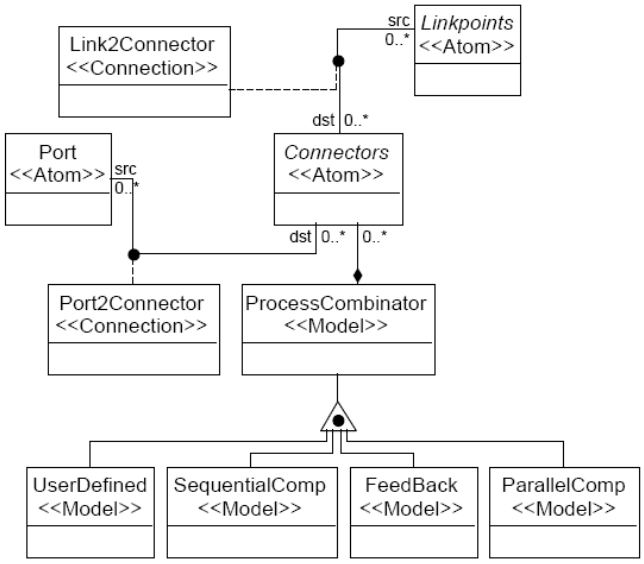


Figure 5.7: Process combinators in GME

The Process Combinator is also an abstract *model* inherited by the combinators defined

in the generic MoCs, which are the sequential, parallel and feedback operators listed in Table 5.1. These enable the modeler to combine processes or PNs in a meaningful manner. We also have an additional operator called *UserDefined*, which can be used to define a new composition operator that the user needs if the basic operators are not sufficient. The functionality of a combinator is to define compositions, therefore they have two streams namely the *Port2Connector* and *Link2Connector*. The *Port2Connector* stream also shown in Figure 5.4 defines the interface of a PN with a combinator, whereas the *Link2Connector* stream also shown in Figure 5.6 defines the interface of a process with a combinator.

5.3 Code Generation Phase

For multi-targeted simulation and verification, we embed the behavioral aspect of the model within the attributes of the model elements during model construction. Consider the example of the adaptive amplifier described in Section 5.4, the output for the current cycle determines the amplifying factor for the next cycle. This behavior is embedded in the attributes of the *scan process element* P_3. These are extracted and tagged in our IML syntax, and furthermore used to populate our *xmlTree*, such that our different print streams can use these embedded behaviors and do a complete translation into executable code for the targeted framework. Furthermore, we perform a partial translation of the structural aspects of the model into a verification template for the SMV model checker. This is a partial template since the translation to a verification environment requires being able to interpret the embedded behaviors, such that the corresponding state machine can be built.

We wrote print streams listed below, which traverse the *xmlTree* extracting info and

translating it to the target framework language allowing the modeler to simulate the model in a multi-targeted environment.

SmlStream: Translates the model into SML that can simulate in the **SML-Sys** framework. The *SmlStream* uses the toplevel model structure from the *xmlTree* to define the structure of the model and simulate it in its respective domain (UMoC/SMoC/TMoC). It also extracts the input and the output signals to the system/model from the toplevel.

Once the target framework and the modeling domain is identified, the *SmlStream* encounters a list of PNs in the *xmlTree*. Translation of a PN structure of the *xmlTree* results in the appropriate definitions of processes, process combinators, PNs and inputs/outputs in the **SML-Sys** framework. Each PN defines a local environment for its components using the SML 'let' construct. The stream translates each PN in the list one after the other until it reaches the end of the list. The end of the list will complete the translation phase.

From the list of PNs, it extracts the first PN and traverses it looking for at most 2 processes or PNs, a combinational operator and its input/output signals. Upon encountering a process structure, it does the following, it extracts the inputs for the process and then extracts the constructor type of this process. During input extraction of a process, the stream extracts the name of the inputs and based on the number of inputs, it creates SML equivalents for these inputs with the extracted names. For the constructor translation, the stream extracts the domain name that indicates the modeling domain of functions to which this constructor should be mapped. Furthermore, it extracts the constructor name, since each constructor is unique, given this name the stream translates it to the respective functions of the corresponding domain in the target framework. Upon identifying the function, the *SmlStream* extracts the arguments for this function. If the argument are empty, meaning that the functionality of this constructor was not embedded during *model construction phase*, then the *SmlStream* inserts comment-lines indicating that the

functionality needs to be added by the user. Upon extracting the attributes, these are translated to arguments for the respective function. Every process has a unique functionality, therefore it is built of a single constructor and these constructors have unique skeletons, therefore translating them to functions in the target platform is performed with relative ease. Upon reaching a composition operator during traversal, it tries to identify whether it is a single-input or double-input operator. A single-input operator corresponds to a feedback composition. If it is a double-input operator, it extracts the name of the operator and its operands, which are processes and PNs that it composes. Once the stream has all the required information regarding the composition operator, it maps the operator to the compositional function in the target framework. Upon encountering another PN, it repeats its traversal step looking for its processes, other PN and the composition operator. If at any point of the translation, the *SmlStream* acquires information that it cannot translate, it will flag an error and the translation will halt.

HaskellStream: Translates the model into Haskell executables that is targeted for the ForSyDe methodology. The *HaskellStream* uses the toplevel model structure of the *xmlTree* to extract the structure of the model and simulate it in the synchronous computational model of ForSyDe. This stream is implemented in a similar fashion as the *SmlStream*. The main distinction between *HaskellStream* and *SmlStream* is that, *SmlStream* translates using eager evaluated semantics, whereas *HaskellStream* follows lazy semantics. In the *HaskellStream*, each PN defines a local environment for the components using the Haskell 'where' clauses.

SmvStream: SMV [16] is a model checker used for verification of hardware systems. It automatically verifies a design for all possible input sequences for properties of combinational logic and interacting finite state machines. We have defined a stream that automatically dumps SMV code for a model. This is a partial translation since the complete translation requires building an engine that deciphers the attributes of the various

components of the model to build the corresponding state machine in SMV. The partial translation that we perform focuses on the structural and communication aspects of the model.

The *SmvStream* uses the toplevel model structure of the *xmlTree* for datatype declaration, which includes the different signals and their domain. These signals are defined using the array construct of SMV. Upon completion of the signal declaration, the *SmvStream* stream encounters a set of PNs. For each of these PNs, the stream creates a module and extracts the inputs, outputs and translates them into SMV code. For all processes within a PN, the stream creates empty module declarations with its inputs and outputs defined from the extraction. A complete translation of a process requires mapping its functionality (specified in SML for the **SML-Sys** framework and in Haskell for ForSyDe) into state machines that has not been implemented. For all other PNs within this PN, the stream repeats the above steps. Finally, the stream extracts the composition operator to map the composition of the different PNs, processes within a PN. It connects the outputs of processes/PNs to the inputs of other processes/PNs based on the composition operator through a sequence of module calls.

Consider a PN example shown in Figure 5.8 with three processes P_1, P_2, P_3 and two PNs PN_1, PN_2. PN_1 is a parallel composition of processes P_1 and P_2 with 2 input signals s_1, s_2 and two outputs s_3, s_4. PN_2 is a sequential composition of PN PN_1 and process p_3 with two input signals s_1, s_2 and an output s_5.

The SMV code illustrating the translation of the PN_2 is shown below:

$$\mathrm{s_3} : \mathrm{P_1(s_1)}$$
$$\mathrm{s_4} : \mathrm{P_2(s_2)}$$
$$\mathrm{s_5} : \mathrm{P_3(s_3.out,\ s_4.out)}$$

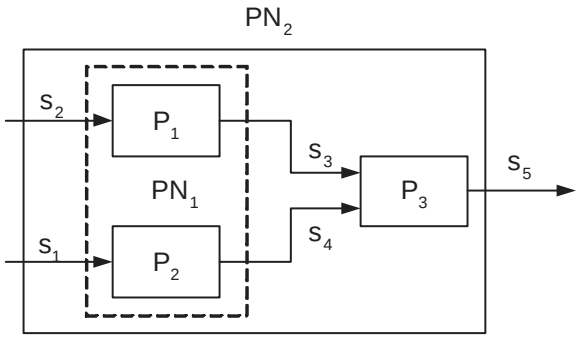


Figure 5.8: A Process Network

where s_3.out and s_4.out are the outputs of process P_1 and P_2.

5.4 Modeling an Adaptive Amplifier

Consider the example of an adaptive amplifier with four processes P_1, P_2, P_3, P_4, two inputs and one output as shown in Figure 5.9. We model this example in the untimed

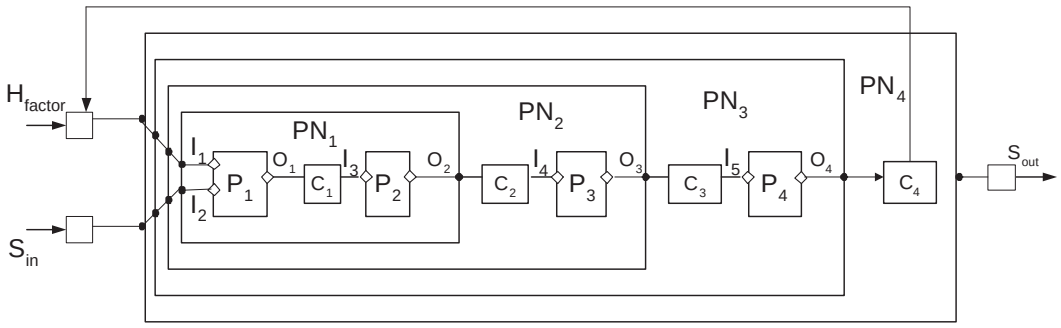


Figure 5.9: Model of an Adaptive Amplifier

domain using our multi-MoC metamodel. We start off by instantiating the UMoC specific metamodel to create an environment for our modeling activity in the untimed domain. We model this as follows - Processes P_1 is a MIP instantiated as a *zip* process, which

takes two inputs S_{in} and H_{factor}. Process P_2 and P_3 are SIPs instantiated as *map* and *scan* processes. Finally, P_4 is an *init* process. The black dots represent the *ports* on the PN and the diamonds are the *links* associated with the processes. The line between a port and a link represents a *Port2Link* connection, whereas the line between two ports represents a *Port2Port* connection. The rectangle between the input/output signal and the port is used to define the *Port2Signal* connection.

P_1 and P_2 are composed using the sequential operator C_1 to form a process network PN_1 as shown in Figure 5.10.

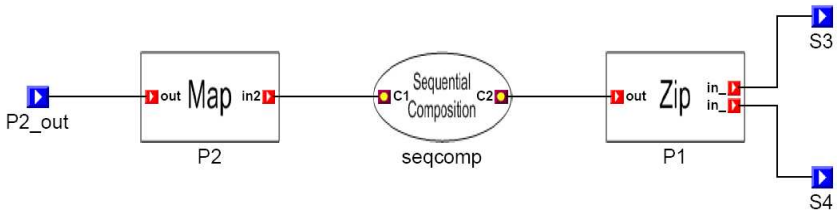


Figure 5.10: PN_1 View

PN_1 is composed using the sequential operator C_2 with P_3 to form PN_2 as shown in Figure 5.11.

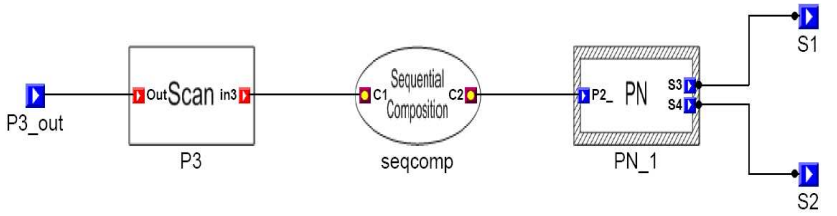


Figure 5.11: PN_2 View

PN_2 is composed using the sequential operator C_3 with P_4 to form PN_3 as shown in Figure 5.12.

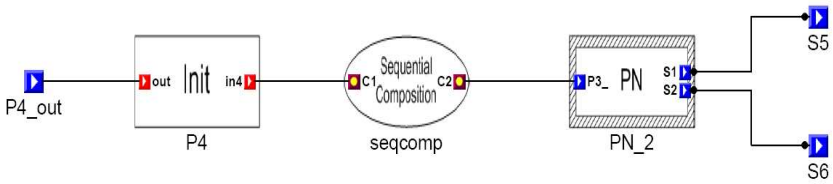


Figure 5.12: PN_3 View

PN_3 is composed with itself in the feedback mode using C_4 as shown in Figure 5.13.

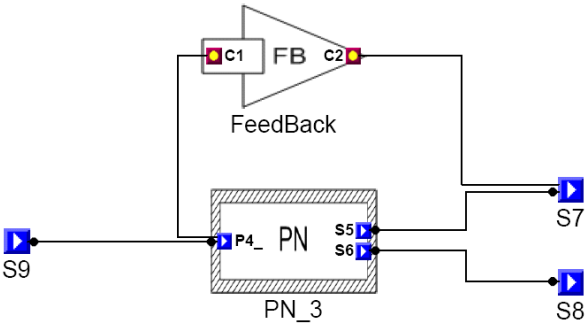


Figure 5.13: PN_4 View

Figure 5.14 shows snapshots of the toplevel of the adaptive amplifier model.

The IML representation for the adaptive amplifier with the outputs of the *SmlStream*, *HaskellStream* and *SmvStream* are shown in Appendix C.

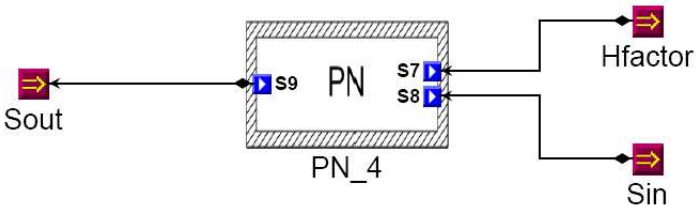


Figure 5.14: Toplevel View

Chapter 6

Intermediate Representation with IML

We discuss the parsing phase of EWD in this chapter. The multi-targeting achieved in EWD is possible through the XML-based interoperability that led to the definition of a platform-independent language called IML. It is a well-defined XML-based modeling syntax for design representation, which has the following advantages:

- Provides interoperability and platform independence.
- Provides extensible modeling syntax.
- Easy to parse, since plenty of XML parsers are freely available.
- Easy to integrate into an alternate design flow.

6.1 Interoperable Modeling Langauge (IML)

IML is an abstract XML-based representation for generic MoCs, which provides a framework-independent language. IML syntax shown in Listing 6.1 is based on the formal semantics of generic MoCs in [8].

Listing 6.1: Snippet of XML syntax

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <!ENTITY % ty_cons "Map | Scan | Mealy | Moore | Zip | UnZip
4  | ZipWith | Scand | Init | Source | Sink">
5
6  <!-- SYNTACTIC CONSTRUCTS FOR THE GENERIC MOC's -->
7  <!ELEMENT MODEL (PROCESS | PN | INPUT | OUTPUT)* >
8  <!ATTLIST MODEL name CDATA #REQUIRED
9        domain CDATA #REQUIRED
10       framework CDATA #IMPLIED >
11
12 <!ELEMENT PN (PN | COMBINATOR | PROCESS | INPUT* | OUTPUT*)+ >
13 <!ATTLIST PN name CDATA #REQUIRED
14         domain CDATA #IMPLIED >
15
16 <!ELEMENT PROCESS (INPUT* | OUTPUT* | CONSTRUCTOR)+ >
17 <!ATTLIST PROCESS name CDATA #REQUIRED
18        domain CDATA #IMPLIED >
19
20 <!ELEMENT CONSTRUCTOR (%ty_cons;)+ >
21 <!ATTLIST CONSTRUCTOR name CDATA #REQUIRED
22             process CDATA #IMPLIED
23             domain CDATA #IMPLIED >
24
25 <!ELEMENT Map EMPTY >
26 <!ATTLIST Map FnUnit CDATA #REQUIRED
27          EventsperCycle CDATA #IMPLIED >
28
29 <!ELEMENT Scan EMPTY >
30 <!ATTLIST Scan EventsperCycle CDATA #IMPLIED
31           NextState CDATA #REQUIRED
```

```
32            InitialState    CDATA #REQUIRED >
33
34  <!ELEMENT Mealy EMPTY >
35
36  <!ATTLIST Mealy EventsperCycle CDATA #IMPLIED
37      NextState       CDATA #REQUIRED
38      OutputFn        CDATA #REQUIRED
39      InitialState    CDATA #REQUIRED >
40
41  <!ELEMENT Moore EMPTY >
42  <!ATTLIST Moore EventsperCycle CDATA #IMPLIED
43                  NextState       CDATA #REQUIRED
44                  OutputFn        CDATA #REQUIRED
45                  InitialState    CDATA #REQUIRED >
46
47  <!ELEMENT Zip EMPTY >
48  <!ATTLIST Zip EventsperCycle_1 CDATA #IMPLIED
49          EventsperCycle_2 CDATA #IMPLIED
50          Size             CDATA #IMPLIED >
51
52  <!ELEMENT ZipWith EMPTY >
53  <!ATTLIST ZipWith EventsperCycle_1 CDATA #IMPLIED
54        EventsperCycle_2 CDATA #IMPLIED
55        FnUnit           CDATA #REQUIRED
56        Size             CDATA #IMPLIED >
57
58  <!ELEMENT UnZip EMPTY >
59  <!ATTLIST UnZip Size CDATA #IMPLIED>
60
61  <!ELEMENT Scand EMPTY >
62  <!ATTLIST Scand EventsperCycle CDATA #IMPLIED
63                  NextState       CDATA #REQUIRED
64                  InitialState    CDATA #REQUIRED >
65
66  <!ELEMENT Source EMPTY >
67  <!ATTLIST Source InitialValue  CDATA #REQUIRED
68                   NextValue     CDATA #REQUIRED >
69
70  <!ELEMENT Sink EMPTY >
71  <!ATTLIST Sink EventsperCycle  CDATA #IMPLIED
72                 OutputFn        CDATA #REQUIRED
```

```
73                  InitialState     CDATA #REQUIRED >
74
75  <!ELEMENT Init EMPTY >
76  <!ATTLIST Init InitialSignal CDATA #REQUIRED >
77
78  <!ELEMENT COMBINATOR EMPTY>
79  <!ATTLIST COMBINATOR name       CDATA #REQUIRED
80            element1 CDATA #REQUIRED
81            element2 CDATA #REQUIRED
82            domain    CDATA #IMPLIED
83
84  <!ELEMENT INPUT EMPTY>
85  <!ATTLIST INPUT name          CDATA #REQUIRED
86       constructor CDATA #IMPLIED
87       value       CDATA #IMPLIED
88       type        CDATA #REQUIRED
89       struct      CDATA #REQUIRED >
90
91  <!ELEMENT OUTPUT EMPTY >
92  <!ATTLIST OUTPUT name          CDATA #REQUIRED
93        constructor CDATA #IMPLIED
94        value       CDATA #IMPLIED
95        type        CDATA #REQUIRED
96        struct      CDATA #REQUIRED >
```

The syntax is built from a set of modeling constructs. It has a toplevel element called **MODEL** that instantiates a model within an MoC based on the values specified for the *domain* and *framework* attributes shown in line 7. A **MODEL** element constitutes of PROCESSes, PNs, INPUTs and OUTPUTs, therefore it can be represented as a 4-tuple grammar G, where $G = \{PN, P, I, O\}$. A **PN** element is built up of a set of processes which compose their inputs and outputs with a set of combinators expressed as $PN = \{PN, P, COMB, I, O\}$ shown in line 12. A **PN** can contain one or more instances of itself, thereby allowing hierarchical PNs. A **PROCESS** element shown in line 16 is defined as $P = \{CONS, I, O\}$ where $CONS$ denotes a process constructor, I denotes a set of input signals and O denotes a set of output signals. A constructor is instantiated by one of

the following: *map*, *scan*, *mealy*, *moore*, *scand*, *zip*, *zipWith*, *unzip*, *source*, *sink* or *init*. In our **SML-Sys** framework, process templates are higher-order functions that support parametric polymorphism. These templates take a set of functions as argument that are mapped to attributes of elements in the IML. These attributes are defined as type CDATA such that they facilitate interoperability by capturing the behaviorial aspect of a model independent of the modeling framework. In line 25 of Listing 6.1, we define the map-based constructor as an entity **Map** with two attributes namely *FnUnit* and *EventsperCycle*. The *FnUnit* attribute contains the functionality of the constructor and the *EventsperCycle* attribute defines the duration in terms of the size of the evaluation cycle.

A **COMBINATOR** element shown in Line 78 defines the compositional operators for our framework. It has a name and domain attribute that identifies the type of the combinator and its domain. It also takes two other attributes that specifies the two processes or PNs that it composes. Finally, IML defines **INPUT** and **OUTPUT** elements shown in Lines 84 and 91. An **INPUT** element defines an input signal as set of attributes which describe the name, domain, type and structure of the signal. Similar is the case with an OUTPUT element.

For the adaptive amplifier example shown in Figure 5.9 of Chapter 5, the toplevel element for its IML representation would hold the following information:

$$M = \{PN, P, I, O\}$$
$$PN = \{PN_1, PN_2, PN_3, PN_4\}$$
$$P = \{P_1, P_2, P_3, P_4\}$$
$$PN_1 = \{\{P_1, P_2\}, C_1, \{S_3, S_4\}, \{P_2.out\}\}$$
$$PN_2 = \{\{PN_1\}, \{P_3\}, C_2, \{S_1, S_2\}, \{P_3.out\}\}$$
$$PN_3 = \{\{PN_2\}, \{P_4\}, C_3, \{S_5, S_6\}, \{P_4.out\}\}$$

$$PN_4 = \{\{PN_3\}, C_4, \{S_7, S_8\}, \{S_9\}\}$$
$$P_1 = \{zipU, \{I_1, I_2\}, \{O_1\}\}$$
$$P_2 = \{mapU, \{I_3\}, \{O_2\}\}$$
$$P_3 = \{scanU, \{I_4\}, \{O_3\}\}$$
$$P_4 = \{initU, \{I_5\}, \{O_4\}\}$$
$$COMB = \{C_1, C_2, C_3, C_4\}$$
$$I = \{H_{factor}, S_{in}\}$$
$$O = \{S_{out}\}$$

Instantiating an IML construct requires giving it a unique occurrence. Since, IML constructors supports hierarchy, elements at different levels can have same names. The instantiation of an IML construct is global and its internals are available to the lower levels using the dot operator, taking advantage of the uniqueness feature of IML. The IML hierarchy is shown in Figure 6.1.

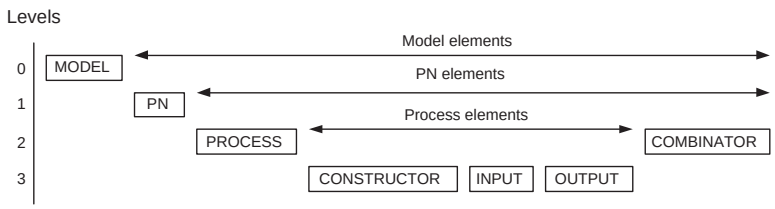


Figure 6.1: The IML Hierarchy

6.2 Parsing Phase

The model and metamodel are stored using XML, which embeds the syntactic constructs of GME in a *.xme* file that needs to be parsed to extract the information specific to our design. We parse the *.xme* file using the **Xerces-C++** parser [1], which is a validating

XML parser and generator written in a portable subset of C++. The parsing is done in two stages, where in the first stage called *Xme2IML*, we parse the .xme file to extract information of the design that is represented in our *IML* syntax stripping out the modeling constructs of GME. This stage is implemented using the SAX (API for XML), where we build an application interface on top of the parser to extract information that can be used to represent the model in our IML syntax. Furthermore, the output of first stage is further parsed in the second stage called *IML2xmlTree* in two phases. In the first phase, a check for *functional errors* like inconsistency and compositional errors is imposed and on verifying the formal correctness of the model it is parsed to populate our *xmlTree* data structure. The second stage of parsing is done by another SAX interface built over the parser to which the output of the first stage is given as input. In the *semantic checking* part of this stage, the model is first checked for correctness in terms of the processes composed for structural equivalence or the output of a process fed as input to another process having a type mismatch. For example an output of a process P_1 which is a signal of integer-valued events sequentially composed to another process P_2 which takes as input a signal of real-valued events. This would flag an exception called compositional errors stating the part of the model that is incorrectly modeled. In the *IML2xmlTree* part of this stage we extract the information pertaining to the modeling hierarchy to populate the *xmlTree*.

6.2.1 Xme2IML

The first stage of the parsing phase analyzes the output of the MC phase to generate the IML representation. The design in GME is exported to an XML-based representation that embeds the model and metamodel-specific information. The user-designed model is wrapped within *Model* tags that have attributes specifying the target framework and the

modeling domain. We extract this information and initialize the **MODEL** construct of the IML. We initialize the **PN** construct by extracting information from the *ProcessNetwork* type tags. The inputs to PNs are extracted from the *InputPort* type tags and similarly the outputs are extracted from the *OutputPort* type tags and used to initialize the **INPUT** and **OUTPUT** constructs for the **PN**. We can extract the constructor-specify information such as name, domain and its arguments from the *SIP*, *MIP* or *PI* type tags and initialize the **PROCESS** construct. The inputs, outputs to processes are extracted from *InputLink* and *OutputLink* tags. The combinator related information can be extracted from the *ProcessCombinator* type tags and used to initialize the **COMBINATOR** construct. The elements that these combinators compose are embedded in the *InputCombinator* and *OutputCombinator* tags.

6.2.2 IML2xmlTree

The second stage of the parsing phase analyzes the IML representation to populate the *xmlTree* structure. The IML has a one to one correspondence with the *xmlTree* structure, therefore the mapping of the IML representation to the *xmlTree* is done with relative ease. The **MODEL** construct is parsed to populate the toplevel of the *xmlTree* structure. The **PNs** are parsed to populate the process network structures. Similarly the **PROCESSes**, **COMBINATORs** are parsed to populate the corresponding structures in the *xmlTree* structure. Finally, the **INPUTs**, **OUTPUTs** are parsed to populate the input, output structures of the different processes, PNs and of the model respectively.

6.2.3 xmlTree Data Structure

The xmlTree consist of a set of classes, where in the toplevel object called *Model* sets the domain, framework and instance name of the model. It is further populated with the set of PNs, processes, inputs and outputs of the model. A PN object is composed of a set of processes, PNs, a combinator, inputs and outputs. Within a PN for each process, we have a process object which inherits a constructor whose functionality is derived from the attributes of the **PROCESS** element of the IML syntax. This feature enables interoperable syntax, since we can have the constructor defined based on the **SML-Sys** Framework or the ForSyDe methodology in [38], allowing the modeler to work with the framework he desires. The combinator object has references to the process or PN elements it composes.

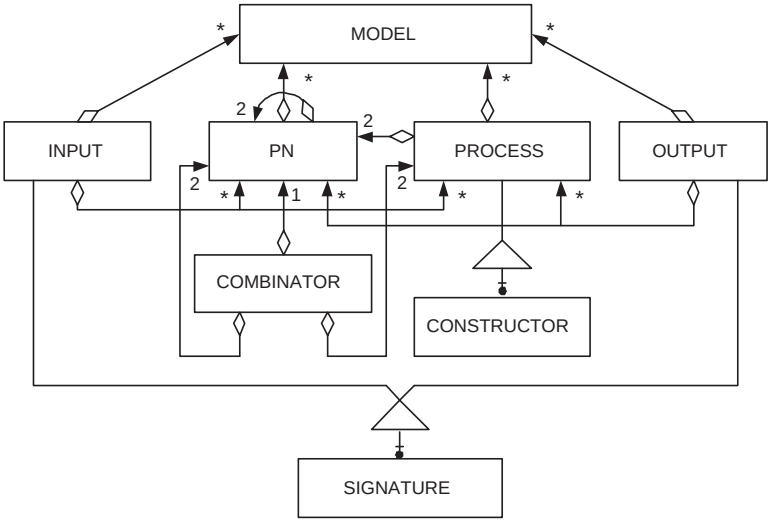


Figure 6.2: The Class Diagram of the *xmlTree* structure

The class diagram for the *xmlTree* data structure is shown in Figure 6.2. Using our *xmlTree* structure instead of the Document Object Manager (DOM) built in by the XML parser provides the tool with semantic error checking capabilities. It also offers the tool the

flexibility to simulate the streams for the model in a multi-targeted environment. It can further facilitates a reverse design flow, where in our structure would be used to generate IML syntax of the model from the target framework.

Chapter 7

Conclusion

Formal verification of a complex model with multiple interacting models of computation is important since interaction between distinct models with different expressibility and features could lead to subtle errors. Therefore, MoCs need to be defined as higher order functions, which provide well-defined formal semantics and facilitate the integration of models of computation, languages and tools on both the syntactic and semantic level. The integration of the UMoC, PSMoC, CSMoC and the TMoC in our **SML-Sys** framework enables the designer to model in a timed, synchronous or untimed fashion, depending on the domain of the problem. Furthermore, they can be used in unison to build heterogeneous models. We also provide a refinement methodology that facilitates the refinement of an abstract specification into an efficient implementation specification by formal design transformations targeting an optimized RTL synthesis. The long term effort is in providing the design automation community with a multi-domain modeling framework based on functional paradigms. The transformation library implemented further facilitates the refinement of the functional specification into an optimized implementation specification which can be used to synthesize an RTL model, thereby reducing

the abstraction gap.

Our work on creating the **EWD** environment and methodology has been motivated by two reasons: (i) the existing multi-MoC modeling frameworks, including Ptolemy, and SystemC extensions, have underlying semantics which do not facilitate formal cross domain analysis, thereby causing some researchers to look into functional programming based multi-MoC frameworks, such as ForSyDe (Haskell based), and **SML-Sys**. (ii) Concise denotational semantic of these frameworks allow functional design and subsequent formal analysis and transformations towards implementation, but require textual programming for system design, leaving designers with the need for a visual environment on top of such a framework. The metamodeling based GME freely available from Vanderbilt not only motivated us to fulfill this need for visual environment on top of functional modeling framework, but also allowed us to rigorously enforce the metamodel of untimed, synchronous, clocked synchronous and timed MoCs, together with various semantic constraints through attributes of these models. GME also helps designers to store executable behaviors in the attributes in the form of code fragments, which can later be used in the body of the generated executable models in SML or Haskell.

Finally, the **EWD** tool requires installation of GME and either the SML multi-MoC libraries or the ForSyDe environment. Also, if SMV models are to be generated, SMV formal verification environment needs to be installed for model checking.

Appendix A

Appendix A

Modeling an Untimed FIR-Filter

A Finite Impulse Response filter is a digital filter in discrete time, that is normally implemented through digital electronic computation. The algorithm is given by a sequence ($h_0, h_1, h_2, ..., h_N$) (the impulse response which, defines the filter's properties), an input x_n and an output y_n shown in Figure A.1.

$$y_n = h_0 x_n + h_1 x_{n-1} \cdots\cdots h_N x_{n-N}$$
$$y_n = \sum_{k=0}^{N} h_k \, x_{n-k}$$

Our Implementation of the FIR shown in Figure A.2 consists of a SOURCE module, a COMPUTATION module and a SINK module which are interfaced with I_1 and I_2.

SOURCE Module: This module extracts a signal of 'c' events from an infinite signal sequence.

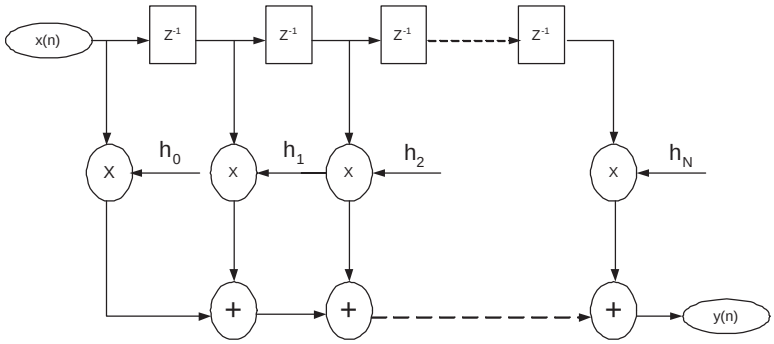


Figure A.1: FIR Block Diagram

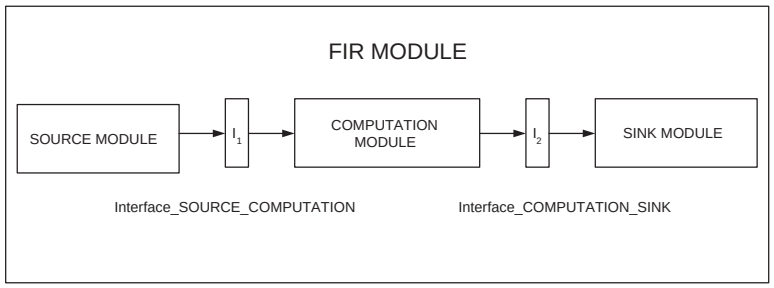


Figure A.2: Implementation of the FIR Block Diagram

COMPUTATION Module: This module constitutes of a convolution function that computes y(k) from the input signal x(k) and the filter's impulse response signal. For each input event, we zip all the previous input events with the respective impulse response coefficients using the zip-based process and then apply a map-based process to compute the convoluted signal.

SINK Module: In this module, we apply the sink-based process constructor to the output of the COMPUTATION module.

Main Module: We define interface functions *Interface_ SOURCE_ COMPUTATION* between SOURCE and COMPUTATION blocks and (Interface_ COMPUTATION_ SINK) between COMPUTATION and SINK blocks. These interfaces are process compositions.

We also implement an FIR module, which determines the specifications for the interfaces such as input size, input rate, output rate and the computational rate for the integration of the model. This implementation of the FIR filter is shown in Listing A.1.

Listing A.1: Model of an FIR Filter

```
1  (* Integrated Module *)
2
3  fun FIR (inp_rate, comp_rate, out_rate, inp_size) =
4  fn (s) => I_COMP_SNK (out_rate)
5  (I_SRC_COMP (inp_rate,comp_rate,inp_size,s,[],0))
6
7    (* Interface between the SOURCE and the COMPUTATION module *)
8    fun I_SRC_COMP (i_rate, c_rate, i_size, i_resp, start, out) =
9    (
10   if i_size = 0  then [ ]
11   else
12   (
13   if start = [] then
14   (COMPUTATION (take (SOURCE(i_size),i_rate),c_rate)
15   imp_resp) @ Interface_SRC_COMP(c_rate,c_rate,(i_size - i_rate),
16   drop(i_resp,i_rate), drop(SOURCE (i_size),i_rate),
17   COMPUTATION(take (SOURCE (i_size),i_rate),c_rate)(i_resp))
18   else
19   constructor ((zipUs (1,1) (take (start, i_rate),i_resp)),
20   i_rate, Convolute, output) @ Interface_SRC_COMP
21   (c_rate, c_rate, (i_size - i_rate),
22   drop (i_resp, i_rate), drop(start,i_rate),
23   last (constructor ((zipUs(1,1) (take (start,i_rate),
24   i_resp)), i_rate, Convolute, out)))))
25
26   (* Interface between the COMPUTATION and the SINK module *)
27   fun I_COMP_SNK (out,o_rate) = SINK (o_rate)(out)
28
29 (* Implementation of the SOURCE *)
30 fun SOURCE(c) = take(sourceU(f1,c1), c2)
31
32 (* Implementation of the COMPUTATION *)
33 fun COMPUTATION(s,c) = fn (imp_resp) => mapU(c,Convolute)
34 (zipUs(1,1) (s,imp_resp))
35
```

```
36 (* Implementation of the SINK *)
37 fun SINK(out_rate) = sinkU(f2,f3,out_rate) output
```

Modeling a Synchronous SOBEL Operator

The Sobel operator performs a 2-D spatial gradient measurement on an image and emphasizes regions of high spatial gradient that correspond to edges. Typically it is used to find the approximate absolute gradient magnitude at each point in an input grayscale image. The operator consists of a pair of 3 x 3 convolution masks (G_x, G_y) as shown below.

-1	0	1
-2	0	2
-1	0	1

1	2	1
0	0	0
-1	-2	-1

These masks are designed to respond maximally to edges running vertically and horizontally relative to the pixel grid, one mask for each of the two perpendicular orientations. The masks can be applied separately to the input image, to produce separate measurements of the gradient component in each orientation. These are then combined together to find the magnitude of the gradient. The gradient magnitude is given by:

$$G^2 = {G_x}^2 + {G_y}^2,$$

Our implementation model shown in Figure A.3 has three main modules which are the SOURCE, the SINK and the COMPUTATION.

SOURCE Module: The *SOURCE* module extracts a signal of 'c' events from an infinite signal sequence.

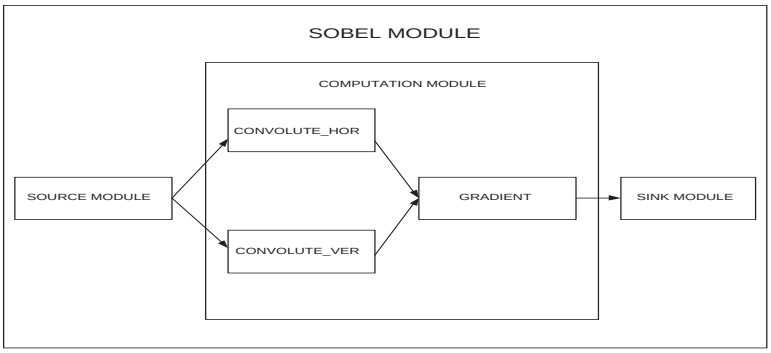


Figure A.3: Implementation of the SOBEL Module

COMPUTATION Module: In the COMPUTATION module, we have a function GRADIENT that computes the magnitude of the gradient with the horizontal orientation component given by the function *CONVOLUTE_HOR* and the vertical orientation component given by the function *CONVOLUTE_VER*. The *CONVOLUTE_VER* and *CONVOLUTE_HOR* functions apply the masks to the input image edges. Computation proceeds by zipping the input edges of image with the respective masks and applying the map-based process to generate the gradient. We have the helper function *getedge* that extracts a part of the image based on the size of the evaluation cycle of the Map based process

SINK Module: In the SINK module, we apply the sink process constructor to the output of the COMPUTATION module.

Main Module: This module determines the specifications for the interfaces such as input size, output rate for the integration of the model. The implementation of the Sobel operator is shown in Listing A.2.

Listing A.2: Model of a Sobel Operator

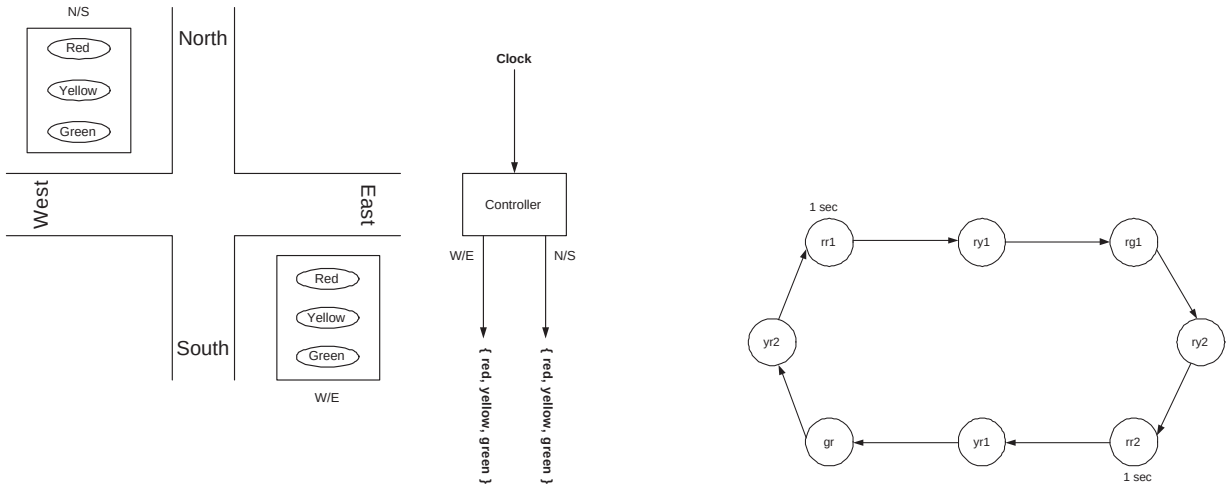
```
1 (* Main Module *)
```

```
2  fun SOBEL(inp_rate,out_rate) =
3  (
4  let                                    (* Definition of bindings *)
5  val inp_edges = SOURCE (inp_rate)   (* Input Edges *)
6  val rsize  = length (inputedges)
7  val csize  = length (head(inp_edges))
8
9  (* Definition of the Horizontal Mask *)
10 val hsobel = [[~1,0,1],[~2,0,2],[~1,0,1]]
11
12 (* Definition of the Vertical Mask *)
13 val vsobel = [[1,2,1],[0,0,0],[~1,~2,~1]]
14
15 val hrow = length (sobel_hopr)
16 val vrow = length (sobel_vopr)
17 val hcol = length (head(sobel_hopr))
18 val vcol = length (head(sobel_vopr))
19 in
20 SINK(out_rate)
21 (GRADIENT
22 (
23  CONVOLUTE_HOR(1,hrow,hcol,rsize,csize,inp_edges,hsobel),
24  CONVOLUTE_HOR(1,hrow,hcol,rsize,csize,inp_edges,vsobel))
25 )
26 end
27 )
28
29 (* Implementation of the SOURCE *)
30 fun SOURCE(c) = take(sourceS(f,c1), c2)
31
32 (* Implementation of the COMPUTATION *)
33
34    (* Implementation of the Horizontal component computation *)
35    fun CONVOLUTE_HOR(_,_,_,_,_,[],_) = []
36    |   CONVOLUTE_HOR(inc,hrow,hcol,rsize,csize,inp_edges,hsobel) =
37    (
38    if csize = inc then
39    mapS(Convolute)(Ziping(getedge(inp_edges,inc,hcol,hrow),hsobel))::
40    CONVOLUTE_HOR(1,hrow,hcol,rsize-1,csize,drop(inp_edges,1),hsobel)
41    else
42    mapS(Convolute)(Ziping(getedge(inp_edges,inc,hcol,hrow),hsobel))::
```

```
43    CONVOLUTE_HOR(inc+1,hrow,hcol,rsize,csize,inp_edges,hsobel)
44    )
45
46    (* Implementation of the Vertical component computation *)
47    fun CONVOLUTE_VER(inc,hrow,hcol,rsize,csize,[],vsobel) = []
48    |   CONVOLUTE_VER(inc,hrow,hcol,rsize,csize,inp_edges,vsobel) =
49    (
50    if col_size = inc then
51    mapS(Convolute)(Ziping(getedge(inp_edges,inc,hcol,hrow),vsobel))::
52    CONVOLUTE_VER(1,hrow,hcol,rsize-1,csize,drop(inp_edges,1),vsobel)
53    else
54    mapS(Convolute)(Ziping(getedge(inp_edges,inc,hcol,hrow),vsobel))::
55    CONVOLUTE_VER(inc+1,hrow,hcol,rsize,csize,inp_edges,vsobel)
56    )
57
58    (* Implementation of the Gradient computation *)
59    fun GRADIENT([],[]) = []
60    |   GRADIENT(h::t,x::y) = [h*h + x*x] @ GRADIENT(t,y)
61
62  (* Implementation of the SINK *)
63  fun SINK(out_rate) = fn (s) => sinkS(f,out_rate) (s)
```

Modeling a Traffic Light Controller (TLC)

Our model of the TLC assumes that there are two traffic lights: one governing the traffic from the North to the South and the other governing the traffic from the East and the West shown in Figure A.4. The controller has a clock signal as its only input, which transmits one event per second. The output controls the colors of the two traffic lights. The duration of the lights are shown in Table A and 1 second for both the lights being red simultaneously. The controller state has two components, one to count the seconds and the other to store the current control signals to the two lights. We denote the first component as *cstate* and the second as *count*. The possible values of cstate with two



(a) A Traffic Light and its Controller (b) State Transition Diagram

Figure A.4: Modeling the Traffic Light Controller Example

Table A.1: Duration of the signals

Color	Duration
Red	68 seconds
Green	60 seconds
Yellow	3 seconds

letters, the first denoting the color to the North-South light and East-West light with the second. The controller is modeled as a mealy state machine and we use the mealyT to instantiate it. The implementation is shown in Listing A.3.

Listing A.3: Model of a Traffic Light Controller

```
1 (* Formulation of a clock as an infinite signal *)
2 val clock_signal = next 1
3
4 datatype 'a seq = nil | cons of 'a * (unit -> 'a seq)
5 fun next (k) = cons(k, fn() => next (k+1))
6
7 (* TLC formulation *)
```

```
8 fun TLC (clock) = mealyT (1,next_state,output,("rr1",0)) (clock)
9
10 fun next_state (clock,("rr1", cnt))
11 = (if cnt < 1  then ("rr1",cnt+1) else ("ry1",0))
12 |   next_state (clock,("ry1", cnt))
13 = (if cnt < 3  then ("ry1",cnt+1) else ("rg",0))
14 |   next_state (clock,("rg", cnt))
15 = (if cnt < 60 then ("rg",cnt+1)  else ("ry2",0))
16 |   next_state (clock,("ry2", cnt))
17 = (if cnt < 3  then ("ry2",cnt+1) else ("rr2",0))
18 |   next_state (clock,("rr2", cnt))
19 = (if cnt < 1  then ("rr2",cnt+1) else ("yr1",0))
20 |   next_state (clock,("yr1", cnt))
21 = (if cnt < 3  then ("yr1",cnt+1) else ("gr",0))
22 |   next_state (clock,("gr", cnt))
23 = (if cnt < 60 then ("gr",cnt+1)  else ("yr2",0))
24 |   next_state (clock,("yr2", cnt))
25 = (if cnt < 3  then ("yr2",cnt+1) else ("rr1",0))
26
27 fun output (clock,("rr1",cnt)) = ["red","red"]
28 |   output (clock,("rr2",cnt)) = ["red","red"]
29 |   output (clock,("ry1",cnt)) = ["red","yellow"]
30 |   output (clock,("ry2",cnt)) = ["red","yellow"]
31 |   output (clock,("rg",cnt))  = ["red","green"]
32 |   output (clock,("yr1",cnt)) = ["yellow","red"]
33 |   output (clock,("yr2",cnt)) = ["yellow","red"]
34 |   output (clock,("gr",cnt))  = ["green","red"]
```

Multi-MoC Example

We discuss the implementation of a Multi-MoC digital equalizer and a verification framework.

Modeling A Digital Equalizer System

We model the digital equalizer system shown in Figure A.5 consisting of a control part, modeled in the Synchronous domain and a dataflow part, modeled in Untimed domain, thereby explaining the interfacing of processes between the UMoC and SMoC domain.

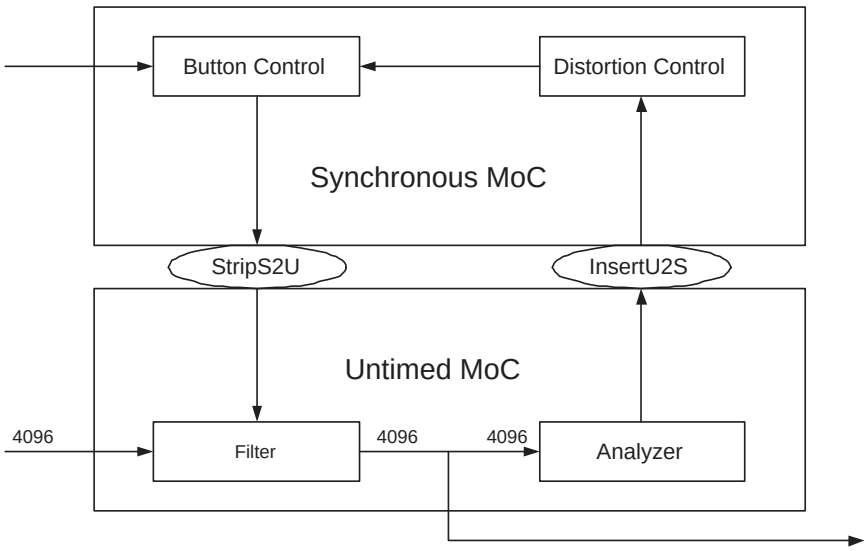


Figure A.5: The Digital Equalizer System

It consist of four processes. The *Filter* process reads the primary input, an audio signal, in chunks of 4096 data points. The filter consists of an amplifier which is controlled by the *Button control process*. Button control receives control input from a user interface, which sets the amplification for volume, bass and the treble of the output audio signal. The output of the filter is analyzed by the *Analyzer process* with the goal of detecting harmful output signals. The *Distortion control process* uses the result of the Analyzer and provides input to the Button control, which in turn sets the parameters for the filter. The digital equalizer has a dataflow and a control part with relatively few control events occurring at irregular points in time. The interface between the Analyzer process and the

Distortion control is implemented as an *insertU2S (interface_ rate)* process, which converts the untimed signal to synchronous signal and the interface between the Button control process and the Filter process is implemented as a *stripS2U* process, which converts the synchronous signal to an untimed control signal. We have a digital_equalizer module shown in Listing A.4, which determines the specifications for the integration such as primary input, control input and the rate of the interface. The functions f_0, f_1, f_2, f_3, f_4 and the constants c_1, c_2, c_3, c_4 are defined with specific functionality for the different processes of the digital equalizer system.

Listing A.4: Implementation of a Digital Equalizer System

```
1  (* The Filter process *)
2  fun Filter (Primary_Input, Control_Input) =
3  UMoC.mapU(c1,f0)(UMoC.zipUs(c1,c2) (Primary_Input,Control_Input))
4
5  (* The Analyzer process *)
6  fun Analyzer (Output) = UMoC.scanU(f1,f2,c3) (Output)
7
8  (* The Distortion Control process *)
9  fun Distortion_control (Analyzed_Output) = SMoC.scanS(f3,c4) (Analyzed_Output)
10
11 (* The Button Control process *)
12 fun Button_control (User_Input, Distortion_Output) =
13 SMoC.mapS(f4) (SMoC.zipS() (User_Input, Distortion_Output))
14
15 (* The Integrated Module *)
16 fun digital_equalizer(Primary_input, Control_input, Interface_rate)
17 = Filter (Primary_input, make_con( length(Primary_input) ))
18 @ filter(Primary_input, stripS2U ()
19 (Button_control (Distortion_control (insertU2S(Interface_rate)
20 (Analyzer (Filter (Primary_input,
21 make_con(length(Primary_input))))))),Control_input)))
```

Modeling a Validation framework

A clocked synchronous process is a perfectly synchronous process that incurred a delay from an input to an output event. Therefore, when we compose a perfectly synchronous process constructor sequentially with a delay process, we get a clocked synchronous process constructor. We formally verify this with a validation model shown in Figure A.6 for a map-based process constructor. The implementation of the validation model is shown in Listing A.5.

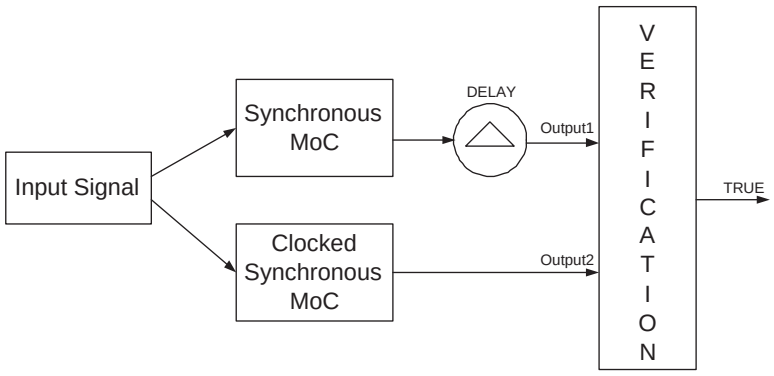


Figure A.6: The Validation Model

Listing A.5: Implementation of a Validation Framework

```
1  (* Synchronous map-based process constructor delayed *)
2  val output1 = CSMoC.delta() (SMoC.mapS(f) (s))
3
4  (* Clocked Synchronous map-based process constructor *)
5  val output2  = CSMoC.mapCS (f) (s)
6
7  (* Comparator *)
8  fun verify ([],[])      = true
9  |   verify ([]._)       = false
10 |   verify (_,[])       = false
11 |   verify (h::t,x::y) = (if h = x then verify(t,y) else false)
```

Appendix B

Appendix B

Domain Interface Constructors

Listing B.1: Domain Interface Constructors

```
1  (* Down_rating Domain Interface constructor *)
2  fun downDI(k) = fn (s) => construct1 (k,s,k)
3
4  fun construct1 (k,[],cnt)   = []
5  |   construct1 (k,h::t,cnt) =
6  (if cnt >= k then [h] @ construct1(k,t,cnt-1)
7  else
8    (if cnt = 0 then [h] @ construct1 (k,t,k-1)
9    else
10   construct1 (k,t,cnt-1)
11   )
12 )
13
14 (* Up_rating Domain Interface constructor *)
15 fun upDI(k) = fn (s) => construct2 (k,s,k)
16
17 fun construct2 (k,[],cnt)   = []
18 |   construct2 (k,h::t,cnt) =
```

```
19  (if cnt = k then [h] @ construct2 (k,t,cnt-1)
20  else
21    (if cnt = 0 then [h] @ construct2 (k,t,k-1)
22    else construct2 (k,h::t,cnt-1)
23    )
24  )
25
26  (* Parallel to Serial Domain Interface constructor *)
27  fun p2sDI(rate) = fn (s) => comp(take(s,rate),f1(s,0),0)
28
29  fun comp (s,cnt,iter) =
30  (if iter = cnt then [] else show (s) @ comp (delete(s),cnt,iter+1))
31
32  fun f1 ([],value) = value
33  |   f1 (h::t,value) =
34  (if value > length(h) then f1(t,value)
35  else f1(t,length(h))
36  )
37
38  fun delete     [] = []
39  |   delete ([]::t) = [] @ delete(t)
40  |   delete (h::t)  = [drop(h,1)] @ delete(t)
41
42  fun show       [] = []
43  |   show ([]::t) = [] @ show(t)
44  |   show (h::t)  = take(h,1) @ show(t)
```

HandshakeWithFIFO Transformation

Listing B.2: The *HandshakeWithFIFO* Rule

```
1   (* The Channel *)
2   fun CIWB(Input,BufferSize) =
3   let
4   fun Channelfn () = fn (sin,sout) =>
5   let
6   datatype SenderState    = SendDataReady|SendData
7   datatype ReceiverState  = WaitReady|WaitData
```

```
8  datatype SenderOutput   = DataReady|Data
9  datatype ReceiverOutput = ProcessInitiator|Ready|Ack
10
11   (* SENDER *)
12   fun Sender() = fn commrec => let
13     val finitequeue : int Queue.queue = Queue.empty
14
15     fun next_u((SendDataReady,q),ProcessInitiator,h) =
16     (SendData,Fifo.enqueue(q,h,BufferSize))
17     |   next_u((SendDataReady,q),_,_) =
18     (SendDataReady,q)
19     |   next_u((SendData,q),Ready,h)  =
20     (SendData,Fifo.enqueue(Fifo.dequeue(q),h,BufferSize))
21     |   next_u((SendData,q),Ack,h)    =
22     (SendData,Fifo.enqueue(Fifo.dequeue(q),h,BufferSize))
23
24     fun out_u((SendDataReady,_)) = [(DataReady,0)]
25     |   out_u((SendData,q))       = [(Data,Fifo.head(q))]
26   in
27     moore2S(next_u,out_u,(SendDataReady,finitequeue)) commrec Input
28   end
29
30   (* RECEIVER *)
31   fun Receiver() = fn comm => let
32
33     fun next_v(WaitReady,(DataReady,_)) = WaitData
34     |   next_v(WaitData, (Data,data))   = WaitData
35
36     fun out_v(WaitReady,(DataReady,_))  = [(Ready,0)]
37     |   out_v(WaitData, (Data,data))    = [(Ack,data)]
38   in
39     mealyS(next_v,out_v,WaitReady) comm
40   end
41
42 val s1 = delayS1(ProcessInitiator) sin
43 val s2 = Sender() s1
44 val s3 = Receiver() s2
45 val (sin,sout) = unzipS2() s3
46 in  (sin,sout) end
47
48 fun fixpoint (f,(v1,v2)) =
```

```
49 let val (v3,v4) = f (v1,v2) in
50 (case equallist (v3,v1) of true => (v3,v4)
51                        | false => fixpoint(f,(v3,[]))) end
52
53 fun FBp(f) = fn s => let val (l3,l4) = fixpoint(f,s)
54 in
55   l4
56 end
57
58 in
59 FBp(Channelfn()) ([],[])
60 end
```

Transformation Rule 1 - MapMerge

Listing B.3: The *MergeMap* Rule

```
1 (* Transformation MapMerge *)
2 fun MapMerge (f,g,signal) = pn_ref (f,g,signal)
3
4 (* Original Process *)
5 fun pn_org (f,g,s) = mapS(f) (mapS(g) (signal))
6
7 (* Transformed Process *)
8 fun pn_ref (f,g,signal) = mapS (combine(f,g)) signal
9 fun combine (f,g) = fn signal => f(g (signal))
```

Transformation Rule 2 - BalancedTree

Listing B.4: The *Balanced* Rule

```
1 (* Transformation Balanced Tree *)
2 fun BalancedTree (f) = fn signal => pn_ref (f) signal
3
```

```
4  (* Original Process *)
5  fun pn_org (g) = fn s =>
6  let
7    val factor = length(s) mod 2
8    val f = func5(g)
9
10   fun func5 (g) = fn s => func6(g,s)
11   fun func6(g,s) =
12   (if length(s) > 1 then func6(g,[zipWithS(g) (first(s),second(s))] @ drop(s,2))
13   else first(s))
14 in
15   (if factor = 0 then zipWithS(f) s else raise Inputs_Not_Powers_of_Two)
16 end
17
18 (* Original Process *)
19 fun pn_ref (f) = fn s =>
20 let
21   val stage = clog(length(s),2)
22   val factor = length(s) mod 2
23
24   fun Merge ([],f) = [] | Merge (list,f) =
25   [f(first(list),second(list))] @ Merge(drop(list,2),f)
26
27   fun BalanceTree(f,s,1) = f(first(s),second(s))
28   |   BalanceTree(f,s,stage) = BalanceTree(f,Merge(s,f),stage-1)
29 in
30   (if factor = 0 then BalanceTree(f,s,stage) else raise Inputs_Not_Powers_of_Two)
31 end
```

Listing B.5: The *PipelinedTree* Rule

Transformation Rule 3 - PipelinedTree

```
1  (* Transformation Pipeline Tree *)
2  fun PipelineTree (f) = fn signal => pn_ref (f) signal
3
4  (* Original Process *)
5  fun pn_org (f) = fn s =>
```

```
6  let
7    val factor = length(s) mod 2
8    val f = func3 (f)
9
10   fun Mg ([],[]) = []
11   |   Mg (list,f1::f2) = [f1(first(list),second(list))] @ Mg(drop(list,2),f2)
12
13   fun Tree(f::[],s,1) = f(first(s),second(s))
14   |   Tree(f,s,stage) = Tree(drop(f,length(s) div 2),
15   Mg(s,take (f,length(s) div 2)),stage-1)
16
17   fun func3(f) = fn s => (if length(s) mod 2 = 0 then Tree(f,s,clog(length(s),2))
18   else raise Inputs_Not_Powers_of_Two)
19 in
20   (if factor = 0 then zipWithS(f) (s) else raise Inputs_Not_Powers_of_Two)
21 end
22
23 (* Original Process *)
24 fun pn_ref (f) = fn s =>
25   let val factor = length(s) mod 2
26   val stage = clog(length(s),2)
27
28   fun Pipeline(f::[],s,1) = transformation3(1,absent_event)(f(first(s),second(s)))
29   |   Pipeline(f,s,stage) = Pipeline(drop(f,length(s) div 2),
30   Merge_With_Delay(s,take(f,length(s) div 2)),stage-1)
31
32   fun transformation3 (k,init) = fn s => Add_delay(k,init,s)
33   fun Add_delay(k,init,[]) = []
34   |   Add_delay(k,init,h::t) = [tag(h)+k::[value(h)]] @ Add_delay(k,init,t)
35
36   fun Merge_With_Delay([],[]) = []
37   |   Merge_With_Delay(list,f::g) = [transformation3(1,absent_event) (f(first(list),
38   second(list)))] @ Merge_With_Delay(drop(list,2),g)
39 in
40   (if factor = 0 then Pipeline(f,s,stage) else raise Inputs_Not_Powers_of_Two)
41 end
```

Transformation Rule 4 - TwoClockDomain

Listing B.6: Implementation of the *TwoClockDomain* transformation

```
1  (* Transformation TwoClockDomain *)
2  fun TwoClockDomain (f,g,s) = fn (s1) => pn_ref (f,g,s) s1
3
4  (* Original Process *)
5  fun pn_org (h,g) = fn (s) =>
6  let val f = func1(h,g) in zipWithS (f) s end
7
8  (* Transformed Process *)
9  fun pn_ref (h,g,(k,value)) = fn (s) =>
10 let
11   val m = length(inputs)
12
13   fun posh(1,x,h::t) = h x
14   |   posh(k,x,h::t) = posh(k-1,x,t)
15
16   fun h_kth(k,x) = posh(k,x,h)
17
18   fun posg(k,x,y,g::t,cnt) =
19   (
20     if cnt = k-1
21     then g(x,y)
22     else posg(k,x,y,t,cnt + 1)
23   )
24
25   fun g_kth(k,x,y) = posg(k,x,y,g,0)
26
27   fun u((0,_),x) = (1,h_kth (1,x))
28     | u((k,value),x) = if k > 0 andalso k < m-1
29     then (k+1,g_kth(k,h_kth(k,x),value))
30     else (0,g_kth(k,h_kth(k,x),value))
31
32   fun v ((k,value)) = (if k = 0 then value else absent_event)
33   fun pfsm (h,g,(k,value)) = fn s => mooreS (u,v,(k,value)) s
34 in
35 downDI(m) (pfsm(h,g,(k,value)) (p2sDI (m) (s)))
36 end
```

Refined Sobel Operator

Listing B.7: Implementation of the Sobel Module

```
1  (* SOBEL OPERATOR *)
2  fun SOBEL_MODULE() = fn inputedges =>
3  let
4
5  (* SOURCE MODULE *)
6  fun SOURCE() = fn inputedges => inputedges
7
8  (* SOBEL CONVOLUTION *)
9  fun SOBEL_OPERATOR() = fn inputedges =>
10 let
11 val sobel_hopr = [~1,~2,~1,0,0,0,1,2,1]
12 val sobel_vopr = [1,0,~1,2,0,~2,1,0,~1]
13
14 fun GRADIENT ([],_) = []  | GRADIENT (edges,mask) =
15 let
16 val rows = length(inputedges)
17
18 fun groupS3(k) = fn inp1 => fn inp2 => fn inp3 =>
19 let
20 fun nt_state(s,x,y,z) = (if length(s) = k orelse length(s) = 0
21 then [x,y,z] else s @ [x,y,z])
22 fun ot_fn(s)      = (if length(s) = k then [s] else [[absent_event]])
23 val f = nt_state
24 val g = ot_fn
25 in
26 mooreS3(f,g,[]) inp1 inp2 inp3
27 end
28
29 fun SumProd ([],[]) = 0 | SumProd (x::xs,y::ys) = x*y + SumProd(xs,ys)
30 fun CONVOLUTE (mask) = fn pts => SumProd(mask,pts)
31 in
32 (
33 if length(edges) >= 3 then
34 let
35 val inp1 = first (edges)
36 val inp2 = second(edges)
37 val inp3 = third (edges)
```

```
38  in
39  (upDI(3,absent_event)
40  (mapS(psi(CONVOLUTE(sobel_hopr)))
41  (downDI(3)
42  (groupS3(9) inp1 inp2 inp3)))
43  )
44  end @ GRADIENT(drop(edges,1),mask)
45  else [] )
46  end
47
48  fun double() = fn h => h * h
49  fun comb(x,y) = [apsi(double()) x + apsi(double()) y]
50  in
51  (mapS(ppsi(Real.round))
52  (mapS(spsi(Math.sqrt))
53  (mapS(rpsi(Real.fromInt))
54  (zipWithS2(comb)
55  (GRADIENT(inputedges,sobel_hopr))
56  (GRADIENT(inputedges,sobel_vopr))))))
57  end
58
59  (* SINK MODULE *)
60  fun SINK() = fn intensities =>
61  let
62  fun f(w) = w
63  in
64  sinkS(f,0) intensities
65  end
66
67  in
68  SINK() (CIWB (SOBEL_OPERATOR() (SOURCE() inputedges),1))
69  end
```

Appendix C

Appendix C

The IML for the Adaptive Amplifier

Listing C.1: The *IML* for the Adaptive Amplifier

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE model SYSTEM "modeling.dtd">
<model name = "Adaptive_Amplifier" domain = "UntimedMoC" framework = "SMLFramework" >

<processnetwork name = "Block3" domain = "UntimedMoC" >
  <processnetwork name = "Block2" domain = "UntimedMoC" >
    <processnetwork name = "Block1" domain = "UntimedMoC" >

      <process name = "P2" >
        <constructor name = "MapS" >
          <Map EventsperCycle = "1" FnUnit = "Undefined" />
        </constructor>
        <input name = "in2" struct = "list" type = "Integer (int)" value = "P1_out1" />
        <output name = "out2" struct = "list" type = "Integer (int)" value = ""/>
      </process>

      <process name = "P1" >
        <constructor name = "ZipS" >
```

```
19            <Zip EventsperCycle_1 = "5" EventsperCycle_2 = "1" Size = "2" />
20          </constructor>
21        <input name = "in_1" struct = "list" type = "Integer (int)" value = "S3" />
22        <output name = "out1" struct = "list" type = "Integer (int)" value = ""/>
23        <input name = "in_2" struct = "list" type = "Integer (int)" value = "S4" />
24        </process>
25
26        <combinator name = "seqcomp" Element1 = "P1" Element2 = "P2" />
27
28        <input name = "S3" struct = "list" type = "Integer (int)" value = "S1" />
29        <input name = "S4" struct = "list" type = "Integer (int)" value = "S2" />
30        <output name = "Block1_out" struct = "list" type = "Integer (int)" value = "P2_out" />
31
32      </processnetwork>
33
34      <process name = "P3" >
35        <constructor name = "ScanS" >
36          <Scan EventsperCycle = "1" InitialState = "Undefined" NextState = "Undefined" />
37        </constructor>
38        <input name = "in3" struct = "list" type = "Integer (int)" value = "Block1.Block1_out" />
39        <output name = "Out3" struct = "list" type = "Integer (int)" value = ""/>
40      </process>
41
42      <combinator name = "seqcomp" Element1 = "Block1" Element2 = "P3" />
43      <input name = "S1" struct = "list" type = "Integer (int)" value = "S5" />
44      <input name = "S2" struct = "list" type = "Integer (int)" value = "S6" />
45      <output name = "Block2_out" struct = "list" type = "Integer (int)" value = "P3_out3" />
46
47    </processnetwork>
48
49    <process name = "P4" >
50      <constructor name = "InitS" >
51        <Init InitialSignal = "[10]" />
52      </constructor>
53
54      <input name = "in4" struct = "list" type = "Integer (int)" value = "Block2.Block2_out" />
55      <output name = "out4" struct = "list" type = "Integer (int)" value = ""/>
56    </process>
57
58  <combinator name = "seqcomp" Element1 = "Block2" Element2 = "P4" />
59  <input name = "S5" struct = "list" type = "Integer (int)" value = "Hfactor" />
```

```
<input name = "S6" struct = "list" type = "Integer (int)" value = "Sin" />
<output name = "Block3_out" struct = "list" type = "Integer (int)" value = "P4_out4" />
</processnetwork>

<output name = "Sout" struct = "list" type = "Integer (int)" value = "Block3.Block3_out" />
<input name = "Hfactor" struct = "list" type = "Integer (int)" value = "[]" />
<input name = "Sin" struct = "list" type = "Integer (int)" value = "[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
</model>
```

The SmlStream for the Adaptive Amplifier

Listing C.2: The *SmlStream* for the Adaptive Amplifier

```
(* Adaptive Amplifier Model *)
structure Adaptive_Amplifier = struct

    (* PN1 *)
    fun Block1() = fn S3 => fn S4 =>
    let

        (* A1 *)
        fun A1() = fn A1_in1 => fn A1_in2 =>
        let
        (The functionality of a,b are defined in the attributes of the object)
        in Zip(a,b) (A1_in1,A1_in2) end

        (* A2 *)
        fun A2() = fn A2_in =>
        let
        (The functionality of c,f are defined in the attributes of the object)
        in Map(c,f) A2_in end

    in seqcomp(A2(),A1() S3) S4 end

    (* PN2 *)
    fun Block2() = fn S1 => fn S2 =>
    let

```

```
26          (* A3 *)
27          fun A3() = fn A3_in =>
28          let
29          (* The functionality of g,f,w are defined in the attributes of the object *)
30          in Scan(g,f,w) A3_in end
31
32      in seqcomp(A3(),Block1() S1) S2 end
33
34      (* PN3 *)
35      fun Block3() = fn S5 => fn S6 =>
36      let
37
38          (* A4 *)
39          fun A4() = fn A4_in =>
40          let
41          (* The functionality of r is defined in the attributes of the object *)
42          in Init(r) A4_in end
43
44      in seqcomp(A4() ,Block2() S5) S6 end
45
46      (* PN4 *)
47      fun Block4() = fn S7 => fn S8 => FBp(Block3())S7
48
49  end
```

The HaskellStream for the Adaptive Amplifier

Listing C.3: The *HaskellStream* for the Adaptive Amplifier

```
1  (* Adaptive Amplifier Model *)
2  module Adaptive_Amplifier where
3
4    (* <process> *)
5    fun P2 in2  = Map in2
6    where
7      (* Define intermediate lambda functions if required *)
8
9    (* <process> *)
```

```
10  fun P1 in_1 in_2  = Zip in_1 in_2
11  where
12    (* Define intermediate lambda functions if required *)
13
14  (* <process> *)
15  fun P3 in3  = Scan in3
16  where
17    (* Define intermediate lambda functions if required *)
18
19  (* <process> *)
20  fun P4 in4  = Init in4
21  where
22    (* Define intermediate lambda functions if required *)
23
24  (* Process Network *)
25  fun Block1 S3 S4  = seqcomp P1 in_1 in_2  P2 in2
26  where
27
28    (* <process> *)
29    fun P2 in2  = Map in2
30    where
31      (* Define intermediate lambda functions if required *)
32
33    (* <process> *)
34    fun P1 in_1 in_2  = Zip in_1 in_2
35    where
36      (* Define intermediate lambda functions if required *)
37
38  (* Process Network *)
39  fun Block2 S1 S2  = seqcomp Block1 S3 S4 P3 in3
40  where
41
42    (* <process> *)
43    fun P3 in3  = Scan in3
44    where
45      (* Define intermediate lambda functions if required *)
46
47  (* Process Network *)
48  fun Block3 S5 S6  = seqcomp Block2 S1 S2 P4 in4
49  where
50
```

```
51      (* <process> *)
52      fun P4 in4  = Init in4
53      where
54        (* Define intermediate lambda functions if required *)
```

The SmvStream for the Adaptive Amplifier

Listing C.4: The *SmvStream* for the Adaptive Amplifier

```
1  /* Adaptive Amplifier Model */
2
3  typedef Integer 0..200;
4  typedef Bool 0..1;
5  typedef state {s0,s1,s2,s3,s4,s5,s6,s7,s8,s9,s10};
6  typedef IntList struct { list  : array 0..4 of Integer; count : Integer; }
7  typedef IntListList struct { list  : array 0..2 of IntList; count : Integer; }
8
9    module P2(in2)
10   {
11   out2 : /* DataType Not Defined */
12   out2 := func.out;
13   }
14
15   module P1(in_1,in_2)
16   {
17   out1 : /* DataType Not Defined */
18   }
19
20   module P3(in3)
21   {
22   Out3 : /* DataType Not Defined */
23   Out3 := stateFunc.out;
24   }
25
26   module P4(in4)
27   {
28   out4 : /* DataType Not Defined */
29   }
```

```
30
31  module Block1(S3,S4)
32  {
33  Block1_out : /* DataType Not Defined */
34  Tout1 : P1(S3,S4);
35  Tout2 : P2(Tout1.out1);
36  Block1_out := Tout2.out2;
37  }
38
39  module Block2(S1,S2)
40  {
41  Block2_out : /* DataType Not Defined */
42  }
43
44  module Block3(S5,S6)
45  {
46  Block3_out : /* DataType Not Defined */
47  Tout1 : Block2(S5,S6);
48  Tout2 : P4(Tout1.Block2_out);
49  Block3_out := Tout2.out4;
50  }
51
52  /* <Main Model> */
53
54  module main()
55  {
56  Hfactor : /* DataType Not Defined */
57  Sin : /* DataType Not Defined */
58  Sout : /* DataType Not Defined */
59
60
61  Fout0 : Block3(Hfactor,Sin);
62  Sout := Fout0.Block3_out;
63  }
```

Bibliography

[1] Apache XML, *Xerces-C++ Website*, http://xml.apache.org/xerces-c/.

[2] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, *Lava: Hardware design in Haskell*, In proceedings of the International Conference of Functional Program, 1998, pp. pp. 174 – 184.

[3] Bluespec Inc, *Bluespec's Website*, http://www.bluespec.com/.

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

[5] National Research Council, *Embedded everywhere*, National Academy Press (2001).

[6] Haskell Group, *Haskell Home Page*, http://www.haskell.org/.

[7] Paul Hudak, *Conception, evolution, and application of functional programming languages*, ACM Computing Surveys **21** (1989), no. 3, pp.359 – 411.

[8] A. Jantsch, *Modeling Embedded Systems and SOC's Concurrency and Time in Models of Computation*, Morgan Kaufmann Publishers, 2003.

[9] L. Lavagno, A. Sangiovanni-Vincentelli, and E. Sentovich, *Models of computation for embedded*, (1998).

[10] A. Ledeczi, M. Maroti, A. Bakay, and G. Karsai, *The Generic Modeling Environment*, Vanderbilt University, Institute for Software Integrated Systems Nashville, 2001.

[11] E. A. Lee and T.M. Parks, *Dataflow process networks*, May 1995, pp. 773 – 799.

[12] E. A. Lee and A. L Sangiovanni-Vincentelli, *Comparing models of computation*, International Conference on Computer-Aided Design (ICCAD), 1996, pp. 234 – 241.

[13] Yanbing Li and Miriam Leeser, *Hml, a novel hardware description language and its translation to vhdl*, IEEE Transaction on Very Large Scale Integration (VLSI) Systems (Vol. 8, ed.), February 2000, p. No. 1.

[14] W. Luk and T.Wu, *Toward a declarative framework for hardware-software*, In proceedings of 3rd International Workshop Hardware/Software Co-design, 1994, pp. pp. 181 – 188.

[15] D. A. Mathaikutty, H. D. Patel, and S. K. Shukla, *A Functional Programming Framework of Heterogeneous Model of Computations for System Design*, FDL, Lille, France, 2004.

[16] K. McMillan, *Symbolic Model Checking*, 1993, Kluwer Academic Publishers.

[17] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The Definition of Standard ML-Revised*, Cambridge MA: MIT Press, 1997.

[18] Gordon Moore, *Cramming more components onto integrated circuits*, **38** (1965), 8.

[19] G. Nordstrom, J. Sztipanovits, G. Karsai, and A. Ledeczi, *Metamodeling - rapid design and evolution of domain-specific modeling environment*, Proceedings of the IEEE,, April 1999, pp. 68–74.

[20] C. Okasaki, *Purely functional data structures*, Cambridge University Press, 1992.

[21] OSCII, *SystemC Website*, http://www.systemc.org/.

[22] H. D. Patel and S. K. Shukla, *SystemC Kernel Extensions for Heterogeneous System Modeling - A Framework for Multi-MoC Modeling & Simulation*, Kluwer Academic Publishers, 2004.

[23] ———, *Towards a Heterogeneous Simulation Kernel for System Level Models: A Systemc Kernel for Synchronous Data Flow Models*, VLSI 2004, 2004.

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

[25] F. Rabhi and G. Lapalme, *Algorithms: A functional programming approach*, Addison-Wesley, 1999.

[26] I. Sander, *System Modeling and Design Refinement in ForSyDe*, Ph.D. thesis, KTH, Sweden, 2003.

[27] I. Sander and A. Jantsch, *System Modeling and Transformational Design Refinement in ForSyDe*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, January 2004.

[28] R. Sharp and A. Mycroft, *A higher level language for hardware synthesis*, In proceedings of the 11th Advanced Research Working Conference on Correct Hardware Design Verification Methods (CHARME) (LNCS Vol. 2144, ed.), 2001, pp. pp. 228 – 243.

[29] D. Shin, S. Abdi, and D. D. Gajski, *Automatic Generation of Bus Functional Models from Transaction level Models*, Proceedings of the Asia and South Pacific Design Automation Conference, Yokohama, January 2004.

[30] System Verilog, *System Verilog Website*, http://www.systemverilog.org/.

[31] W. Taha, P. Hudak, and Z. Wan, *Directions in Functional Programming for Real(-Time) Applications*, Yale University.

[32] J. P. Talpin, P. L. Guernic, S. K. Shukla, and F. Doucet, *Formal Refinement Checking in a System-level Design Methodology*, Fundamenta Informaticae, 2004.

[33] S. Thompson, *Haskell - The Craft of Functional Programming, 2 ed*, MA: Addison-Wesley, 1999.

[34] University of California at Irvine, *SpecC Website*, http://www.ics.uci.edu/ specc/.

[35] Vanderbilt University, *GME Website*, http://www.isis.vanderbilt.edu/Projects/gme.

[36] VERILOG , *VERILOG Website*, http://www.verilog.com/.

[37] VHDL, *VHDL Website*, http://www.vhdl.org/.

[38] W. Wu, I. Sander, and A. Jantsch, *Transformational system design based on a formal computational model and skeletons*, FDL, Tubingen, Germany, September 2000.

Vita

Deepak Abraham Mathaikutty

Date of Birth	:	September 5, 1981
Marital Status	:	Single
Visa Status	:	F-1

Office Address:

FERMAT Research Lab.
340 Whittemore Hall
Blacksburg, VA 24060
Email: mathaikutty@vt.edu
URL: http://www.fermat.ece.vt.edu/
Tel: (540) 443-1315

Research Interests

(a) Formal Methods

(b) Embedded Systems Design

(c) Hardware & Software Co-Design Methodologies

(d) System Level Design and Validation

(e) Models of Computation & Multi-MoC Modeling

(f) Functional Modeling Frameworks & MetaModeling

Education

(b) M.S., Computer Engineering, (Expected May 2005), Virginia Polytechnic Institute and State University.
M.S. Thesis Title: "Functional Programming and Metamodeling frameworks for System Design"

(c) B.S., Computer Science and Engineering, (May 2003), National Institute of Technology (REC), Trichy.

Current Work

(a) SML-Sys: A Multi-MoC functional framework for System Design
Implemented in a functional language Standard ML based on functional paradigms which facilitates the application of formal methods for transformation, synthesis and verification.

(b) Correctness Preserving Design Refinements in a Functional Framework for System Design
A refinement based design providing formal transformation methods for a transparent refinement process to an optimized implementation specification that can be used for RTL synthesis.

(c) EWD: A Metamodeling Driven Customizable Multi-MoC System Modeling Environment
A visual modeling environment for multi-MoC modeling of embedded software and hardware systems.

(d) SystemCXML: A Project developed to retrieve Structural Information from SystemC Models
Developed an innovative way to parse SystemC using Doxygen and XML utilities. To be uploaded at http://systemcxml.sourceforge.net.

Skills

– Languages: C, C++, SystemC, Verilog, VHDL, SML, Haskell, Matlab, LISP, PROLOG, PERL, XML, Intel/Motorola Assembly

– Environments: Windows 95/ 98/NT/2000/XP, DOS, Linux.

Experience

(a) January 2004 to present: Graduate Research Assistant, Virginia Tech., Blacksburg, VA.

- Functional Programming Framework for Heterogeneous Models of Computation for System Design.
- Correctness Preserving Design Refinements in a Functional Programming Framework for System Design.
- A Multi-Target Modeling Environment For Multi-MoC System Models with Meta-Modeling, XML, and Functional Paradigms.
- Investigated EDG C++ Front-end parsing for SystemC. After some effort, we resorted to using Doxygen along with XML utilities to construct the FERMAT's SystemC parser.
- Investigating CARH, a service oriented architecture for validation and verification. Adding services for better model visualization, tracing, and debugging capabilities.

(b) June 2003 to December 2003: Technical Trainee in FirstApex, Bangalore (Apex Technologies Pvt. LTD, India).

- Design and Deployment of a mod for Xgen, FirstApex's general insurance software product.
- Design of data collection Tools to query and generate Reports on a set of parameters from their Insurance Domain
- Design and Deployment of a Time Management System (TIMAN), for FirstApex Bangalore.

[1] D. Bhaduri, D. Mathaikutty, and S. Shukla, *An automated reliability enhancement tool for defect-tolerance*, Tech. Report 2004-16, Virginia Tech. FERMAT Lab, 2004.

[2] H. Patel, D. Mathaikutty, and S. Shukla, *Carh: A service oriented architecture for dynamic verification of systemc models*, Tech. Report 2004-19, Virginia Tech. FERMAT Lab, 2004.

[3] S. Suhaib, D. Mathaikutty, D. Berner, and S. Shukla, *Extreme formal modeling for hardware models*, 5th International Workshop on Microprocessor Test and Verification (MTV'04), September 2004.

[4] S. Suhaib, D. Mathaikutty, and S. Shukla, *Metamodeling based modeling and verification flow for concurrent applications using multi-threaded graphs*, Tech. Report 2004-09, Virginia Tech. FERMAT Lab, 2004.

[5] D. Mathaikutty, H. Patel, and S. Shukla, *Correctness preserving design refinements in a functional programming framework for system design*, Tech. Report 2004-07, Virginia Tech. FERMAT Lab, 2004.

[6] ———, *A functional programming framework of heterogeneous model of computation for system design*, Forum of Specification and Design Languages (FDL'04), September 2004.

[7] D. Mathaikutty, H. Patel, S. Shukla, and A. Jantsch, *Ewd: A metamodeling driven customizable multi-moc system modeling environment*, Tech. Report 2004-20, Virginia Tech. FERMAT Lab, 2004.

[8] D. Mathaikutty, S. Suhaib, and S. Shukla, *Effects of property ordering in an incremental formal modeling methodology*, IEEE International High Level Design Validation and Test Workshop (HLDVT'04), November 2004.

[9] ———, *Property ordering effects in an incremental formal modeling methodology*, Thirteenth International Workshop on Logic and Synthesis (IWLS'04), June 2004.

Selected Projects

(a) **CycleComputer**: A hardware prototype of a complete bicycle computer simulated and synthesized

in verilog.

(b) Developed an **Integrated Firewall Toolkit** which filters packets, performs authentication and verification services.

(c) Design and Implementation of the a **Generalized Protocol Tunneling**, which facilitates different networks to communicate and transfer data without worrying about the underlying mismatch in protocols.

(d) Developed a Simulation Model in C for the **Admission Control Policy** for new and handoff calls in mobile networks and providing a threshold bandwidth management mechanism.

(e) Developed a **Protocol combining the principles of RSVP and Active Networks** to provide support of QoS for multimedia application over mobile networks.

(f) Developed a **Workbench to assist Transliteration and Font deciphering** for Indian languages (NLP) by defining predicates to do a character mapping.

(g) Implemented a **File Transfer Protocol Client** Using UNIX C providing a GUI to the FTP client so that it is more user friendly.

References

(1) Professor Sandeep K. Shukla
Bradley Department of Electrical and Computer Engineering,
Virginia Polytechnic Institute and State University
340 Whittemore Hall
Blacksburg, VA 24061
Tel: (540) 231-2133
Fax: (540) 231-3362
Email: shukla@vt.edu

(2) Professor Rajib Mall
Department of Computer Science and Engineering,
Indian Institute of Technology
Kharagpur 721 302, India
Tel: (03222) 83482
Fax: (03222) 55303
Email: rajib@cse.iitkgp.ernet.in

(3) Professor K. Viswanathan Iyer
Head of the Computer Science and Engineering Department,
National Institute of Technology,
Tiruchirappalli 620015, India
Tel: (0431) 500281
Fax: (0431) 500133
Email: kviyer@nitt.ernet.in