EWD: A Metamodeling Driven Customizable Multi-MoC System Modeling Framework

DEEPAK MATHAIKUTTY, HIREN PATEL, and SANDEEP SHUKLA
Fermat Lab., Virginia Tech
and
AXEL JANTSCH
Royal Institute of Technology, Sweden

We present the EWD design environment and methodology, a modeling and simulation framework suited for complex and heterogeneous embedded systems with varying degrees of expressibility and modeling fidelity. This environment promotes the use of multiple models of computation (MoCs) to support heterogeneity and metamodeling for conformance tests of syntactic and static semantics during the process of modeling. Therefore, EWD is a multiple MoC modeling and simulation framework that ensures conformance of the MoC formalisms during model construction using a metamodeling approach. In addition, EWD provides a suite of translation tools that generate executable models for two simulation frameworks to demonstrate its language-independent modeling framework. The EWD methodology uses the Generic Modeling Environment for customization of the MoC-specific modeling syntax into a visual representation. To embed the execution semantics of the MoCs into the models, we have built parsing and translation tools that leverage an XML-based interoperability language. This interoperability language is then translated into executable Standard ML or Haskell models that can also be analyzed by existing simulation frameworks such as SML-Sys or ForSyDe. In summary, EWD is a metamodeling driven multitarget design environment with multi-MoC modeling capability.

Categories and Subject Descriptors: C.5 [**Computer System Implementation**]; I.6.5 [**Simulation and Modeling**]: Model Development; D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; J.6 [**Computer-Aided Engineering**]

General Terms: Design, Languages, Verification

Additional Key Words and Phrases: Metamodeling, metamodel, MoC, functional language, denotational semantics, interoperable modeling language, heterogeneous system design, Ptolemy II, SystemC, ForSyDe

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

This work has been supported by the NSF project CCR-0237947 and SRC Integrated Systems Grant.

Authors' addresses: D. Mathaikutty, H. Patel, and S. Shukla, Fermat Lab, Virginia Tech, Blacksburg, VA; email: Shukla@vt.edu; A. Jantsch, Department of Microelectronics and Information Technology, Royal Institute of Technology, Sweden.

ACM Reference Format:
Mathaikutty, D., Patel, H., Shukla, S., and Jantsch, A. 2007. EWD: A metamodeling driven customizable multi-MoC system modeling framework. ACM Trans. Des. Autom. Electron. Syst. 12, 3, Article 33 (August 2007), 43 pages. DOI = 10.1145/1255456.1255470 http://doi.acm.org/10.1145/1255456.1255470

1. INTRODUCTION

Most system models for System-on-Chip (SoC) are heterogeneous and encompass multiple models of computation (MoC)s in their different components. This indicates an essential need for multi-MoC support at various abstraction levels in modeling and simulation frameworks. Furthermore, having a multi-MoC framework is quite useful as an experimentation platform for researchers to integrate different modeling domains and paradigms as well as various design and verification tools. System-level design environments such as Ptolemy II [Brooks et al. 2005] and SystemCH [Patel and Shukla 2005] offer heterogeneous modeling support through their multi-MoC language. However, none of these design environments support easy customization. The customizability aspect of a design environment allows the user to make additions to its modeling capability by introducing support for newer MoCs for specific application domains, which in turn increases the *modeling fidelity*. Informally, fidelity indicates the number of different MoC capabilities supported by a modeling framework.

The capability of a design environment to customize its modeling language is called *metamodeling*. The component of a design environment that allows such a capability is called a *metamodeling framework*. GME [Ledeczi et al. 2001] is an example of such a metamodeling framework that allows the description of a modeling domain by capturing the domain-specific syntax and static semantics into visual notations and rules called the *metamodel*. The dynamic or execution semantics of the modeling domain is provided through interpreters, which analyze and translate the visual models into executables. The metamodeling capability is provided through a set of generic concepts implemented as UML-class diagrams and OCL constraints, resulting in visualization, design-time checks, and customization. We present EWD, a multiple MoC modeling and simulation framework that uses this capability to describe a metamodeling driven multi-MoC modeling formalism. The MoC-specific syntax and static semantics[1] in EWD are expressed with the generic concepts in GME by creating metamodels for them. In this article, we briefly compare EWD against contemporary design environments such as Ptolemy II, SystemCH [Patel and Shukla 2005] and Metropolis [The Metropolis Project Team 2004] with respect to their metamodeling-based customizability.

Simulation frameworks execute the heterogeneous design description through the underlying imperative language-based compilers such as the C++/Java compiler for SystemCH [Patel and Shukla 2005] and Ptolemy II-based designs. Some other simulation frameworks follow an alternative approach of using a functional language, such as the hardware description language

[1]We mean the well-formedness of constructs in the modeling language.

HML [Li and Leeser 2000] implemented in Standard ML (SML) [Milner et al. 1997], which combines strong typing with polymorphism and automatic type inference to express the functionality of the hardware specified. Some MoC-specific functional frameworks are ForSyDe [Sander and Jantsch 2004] implemented in Haskell [Thompson 1999], and the SML-Sys [Mathaikutty 2005; Mathaikutty et al. 2004b] framework in SML. These frameworks are based on formal semantics and functional paradigms which facilitate the application of formal methods for transformation, synthesis, and verification. Furthermore, these frameworks implement MoCs as higher order functions, resulting in a formal underpinning due to their denotational nature.

SML-Sys facilitates the integration of modeling domains, languages, and tools on both the syntactic and semantic level. The uniform representation of different MoCs in the same semantic framework paves the way for analysis, verification, and optimization across MoC domain boundaries. In Jantsch [2003], there is a discription of how functionality can be moved from one MoC domain into another while preserving the system behavior and important properties of the system. Even though we have not yet demonstrated the full potential of a uniform semantic framework that encompasses various MoCs, we believe it will greatly help in the understanding of heterogeneous systems well beyond what pure simulation-based environments and frameworks can accomplish. Note, that we do not require that the designer uses a single, all-encompassing design language to express all different MoCs. Current design languages such as Verilog, VHDL, C, or SystemC can be used to model different parts. We only require that these different parts can be projected onto specific MoCs in our framework. This can be enforced by establishing coding and interpretation rules for each design language. Then the analysis, verification, and optimization techniques of the multi-MoC framework become valid for the models of the frontend design languages. Note, that arbitrary VHDL or SystemC code does not automatically adhere to any specific MoC other than the native MoC of that language. Consequently designers have to be aware of the MoC they are using just like they must adhere to the modeling rules when using VHDL for RTL synthesis.

EWD integrates the SML-Sys framework into the design environment and eases an automatic translation of a high-level MoC-based system description into executable models in SML. SML-Sys is one of the many targets for code generation in EWD. Moreover, we illustrate a metamodeling-based design environment by building metamodels to support the same MoCs used in SML-Sys. Therefore, we have elaborated on SML-Sys, its choice of MoC classification, and its comparison with Tagged Signal Model (Appendix A). Although with the instantiation of a very different metamodel, EWD could be used to enforce various MoCs and MoC interactions, this article concentrates only on metamodel instantiations related to SML-Sys. Therefore, we believe this extra information will help readers in understanding the article better.

The rest of the article is organized as follows: In Sections 2 and 3, we present the necessary background and related work. Section 4 discusses the design-flow of the EWD methodology. In Section 5, we explain the model construction phase of the EWD design flow and use the example of an adaptive amplifier

to illustrate it. In Section 6, we discuss the parsing phase by briefly introducing the IML syntax and intermediate parsing stages. In Section 7, we briefly outline the multitargeted capability that we achieve in EWD, and the following section discusses our experience with EWD and presents the modeling of a digital equalizer example. Finally, we conclude by summarizing the article and discussing the work in progress.

1.1 Main Contributions

EWD's design environment has three main components, a metamodeling framework based on GME, a multi-MoC modeling framework based on the definition of generic MoCs in Jantsch [2003], and simulation frameworks such as SML-Sys and ForSyDe based on functional languages. These three components together provide the following features in EWD:

—a modeling framework that supports visual multi-MoC modeling and enforces semantic constraints during model construction for conformance to the underlying MoC;

—a visual metamodeling framework that provides extendibility through metamodels, allowing the user to enhance/restrict the modeling framework based on the levels of modeling fidelity versus genericity required;

—executable simulation models are automatically built by the tool, and currently the target simulation languages are SML and Haskell-based modeling extensions, namely, SML-Sys and ForSyDe;

—the multitargeting is achieved through XML-based interoperability that led to the definition of an interoperable language and its binding to various target languages.

Furthermore, we provide an extensive comparison of EWD with contemporary design environments such as Ptolemy, Metropolis, and ForSyDe based on their metamodeling and multi-MoC capability. Finally, we provide a case study and outline our design experience from using the EWD environment.

2. BACKGROUND

A model of computation [Lavagno et al. 1998] describes MoC behavior in terms of how the communication proceeds and how the computation occurs. Examples of MoCs are discrete event (DE), synchronous dataflow (SDF), and finite state machine (FSM). A *multi-MoC modeling framework* provides the designer with a language to describe the structure and behavior of systems using MoC-specific constructs. Usually, *languages* such as C++ for SystemC, Java for Ptolemy and UML for EWD, provide support for MoC capabilities in a particular modeling framework. The *modeling fidelity* of a framework is defined as the degree to which a framework supports modeling designs represented by their most natural MoCs. The *expressiveness* of a framework is the overall capability of describing any behavior. A framework that provides constructs for describing the structure and behavior of MoCs has higher modeling fidelity as opposed to one that does not support such constructs. For example, SystemC inherently

supports a DE-based modeling language, but with the SDF extension provided in SystemCH [Patel and Shukla 2005], the designer has additional constructs to easily model SDF designs. This adds support for multi-MoC capability in the system-level design language. Often SystemC designers devise techniques using events to mimic the design's inherent MoC with the side effect of programming overhead, complexity, and the higher possibility of design errors. This shows that the expressiveness of SystemC is also high, providing designers with the capability to describe any behavior using C++. On the other hand, the advantage of representing designs via MoCs is in the reduction of these modeling and design errors as well as reducing the complexity in constructing such designs.

2.1 MoC Classifications

The most well-known classification of MoCs has been stated in the context of Ptolemy II projects [Brooks et al. 2005] called *modeling domains*. Some of Ptolemy II's modeling domains [Lee and Sangiovanni-Vincentelli 1998] are FSM, DE, Continuous Time (CT), Communicating Sequential Processes (CSP), Kahn Process Network (KPN), etc. Another classification of MoCs called *generic MoCs* has been done by abstracting the time of complex designs in the context of ForSyDe project [Sander and Jantsch 2004]. This work can be distinguished from Ptolemy's work as a distinction of the denotational view versus operational view of MoCs. The denotational view of an MoC consists of a recursive formalism of the MoC semantics using mathematical objects. The operational view of an MoC describes a specific implementation of the MoC syntax and semantics based on how the computation takes place and how the communication proceeds.

2.2 Generic MoCs

We briefly introduce the generic MoCs defined in Jantsch [2003]. These MoCs are built on processes, events, and signals. *Events* are the elementary units of information exchanged between processes. *Signals* are finite or infinite sequences of events. The activity of processes is divided into *evaluation cycles*. A *process* partitions its input and output signals into subsequences corresponding to its evaluation cycles. An evaluation cycle defines the number of events in a subsequence. During each evaluation cycle, a process consumes exactly one subsequence of each of its input signals. To relate functions on events to processes, we introduce process constructors. These are parameterizable templates that instantiate processes of a specific computational behavior. Furthermore, we define process combinators to construct *process networks* (PN)s through process compositions.

A *generic MoC* is defined as a set of processes and process networks that are constructed from the given set of process constructors and combinators. It is further categorized 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. The classification of MoCs [Jantsch

2003], characterized by the duration of their evaluation cycle, are: Untimed MoC (UMoC), Synchronous MoC (SMoC), and Timed MoC (TMoC).

(1) *UMoC.* Processes communicate and synchronize based on the order of events in the absence of time.
(2) *SMoC.* The time line is abstracted into uniform intervals. Every computation within an interval occurs at the same time, but the intervals are totally ordered along the timeline, implying the evaluation cycle of processes lasts exactly one time interval. It is further categorized into two, which is based on whether the output event of a process occurs in the same time interval as the corresponding input event (perfectly synchronous MoC) or whether every process incurs a delay from an input event to an output event (clocked synchronous MoC).
(3) *TMoC.* This MoC is a generalization of SMoC. Timing information is conveyed on the signals by transmitting absent events ($\sqcup$) 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.

2.3 SML-Sys Framework

The SML-Sys library facilitates the instantiation of the untimed, synchronous and timed MoCs. Modeling using SML-Sys framework starts by capturing the system behavior into an abstract functional specification. This is refined inside the functional domain by a stepwise application of well-defined transformation rules into an efficient optimized implementation specification. We discuss the UMoC of the SML-Sys framework by introducing the notations and briefly describing the implementation. A process communicates with another process by writing to and reading from signals. The set of values V represents the data[2] communicated over a signal, and the set E constitutes the events that are basic elements of a signal containing values. Untimed events are values and their set is denoted by $\dot{E} = V$ with $\dot{e} \in \dot{E}$ denoting an event. Signals are an ordered sequence of events such that $\dot{e}_i$ denotes the ith event in a signal. We use $\dot{S}$ to denote the untimed signal set and $\dot{s}$ to designate an individual untimed signal. Processes are defined as functions on signals that are mappings between signal sets ($p : \dot{S} \rightarrow \dot{S}$). These mappings are implemented by operating on subsequences instead of on the whole sequence. These operations may have no state (identical subsequences are mapped into identical subsequences) or may have state (identical subsequences may be mapped to different subsequences).

2.3.1 *UMoC in SML-Sys.* UMoC adopts the simplest timing model corresponding to the causality abstraction. Processes modeled as state machines are connected to each other via signals. Signals transport data values from

[2]The data can be any abstract or concrete type.

a sending process to a receiving process. The data values do not carry time information, but the signals preserve the order of emission.

Process Constructors. In the UMoC, we implement a set of process constructors (suffixed with U) that are used to define different computational blocks that are either processes or PNs. We discuss the formulation of a single input-output Mealy-based process constructor with respect to finite signals in the following.

Mealy-based process constructor. This is a constructor that creates a process p with a state, which, when given an input signal $\dot{s}$, generates an output signal $\dot{s}'$. It resembles a Mealy state machine, which is described with four parameters:

(1) ω_0: the initial state of the Mealy state machine and ω_i is the state at the ith evaluation cycle;
(2) f: the output encoding function computes the output subsequence $\dot{a}'_i$ based on the current state ω_i and the corresponding input subsequence $\dot{a}_i$;
(3) g: the next-state function determines the next state based on the current state ω_i and the corresponding input subsequence $\dot{a}_i$;
(4) γ: determines the size of each input subsequence. It computes these sizes based on the current state (ω_i).

The function-based semantics for the *mealyU* process constructor are:

$$\begin{aligned} mealyU(\gamma, g, f, \omega_0) = p \quad & where, \\ p(\dot{s}) &= \dot{s}' \\ \psi(v, \dot{s}) &= \dot{a}_i \\ v(i) &= \gamma(\omega_i) \\ g(\omega_i, \dot{a}_i) &= \omega_{i+1} \\ f(\omega_i, \dot{a}_i) &= \dot{a}'_i \end{aligned}$$

$$\dot{s}, \dot{s}', \dot{a}_i, \dot{a}'_i \in \dot{S},\ \omega_i \in \dot{E},\ i \in \mathbb{N},\ where$$

ψ is the partition function that creates subsequences of the signal $\dot{s}$ based on v.

The list of elementary process constructors implemented in SML-Sys for the UMoC is shown in Table I. The Mealy-based process constructor discussed can be extended to handle multiple inputs making it more generic, which simplifies Table I.

Process Combinators. We define composition operators to combine different processes to form processes and PNs. The sequential, parallel, and feedback operators shown in Figure 1 and the formulation of the feedback operator is as follows.

Feedback composition operator. Given a process $p : (\dot{S} \times \dot{S}) \rightarrow (\dot{S} \times \dot{S})$ with two input signals and two output signals, we define the process $FB_p(p) : \dot{S} \rightarrow \dot{S}$. The behavior of the process $FB_p(p)$ is defined by the *least-fixed point* (LFP) semantics.

$$FB_p(p)(\dot{s_1}) = \dot{s_2}, \text{where} p(\dot{s_1}, \dot{s_3}) = (\dot{s_2}, \dot{s_3}).$$

The output $\dot{s_3}$ is fed back as input to the process and the constructor terminates when the output $\dot{s_3}$ does not change and has fixed pointed. The current

Table I. Process Constructors for the Untimed MoC

Name	UmoC	Description
Map	mapU	Perform a computation 'f' on 'c' events
Scan	scanU	Processes with an internal state and a next state function
Moore	mooreU	Process with a state and the output is a function of the state
Mealy	mealyU	Process with a state and the output depends on the state & current input
Scand	scandU	Scan process with the initial state
Zip	zipU	Zips the inputs together based on two functions
Zip	zipUs	Zips the inputs together based on two constants
Zip	zipWithU	Zips the inputs with an arbitrary function
Unzip	unzipU	Unzips a zipped signal appropriately
Source	sourceU	Initialize a signal with events
Sink	sinkU	Initialize a signal to an empty signal
Init	initU	Initialize a signal with another signal

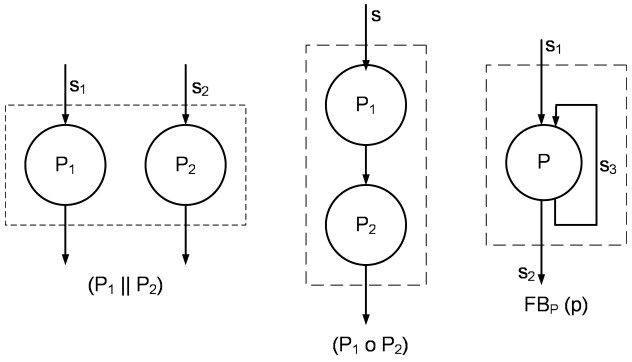


Fig. 1. Parallel, sequential, and feedback operators.

output for $\dot{s_3}$ is computed based on the current input s_1 and the previous output of $\dot{s_3}$ (given as input). The process $FB_p(p)$ terminates when the current output $\dot{s_3}$ and the previous input $\dot{s_3}$ is the same, implying that a pattern is repeating on $\dot{s_3}$, which is the fix-point value of the process.

Finally, the *untimed model of computation* is defined as UMoC = (C,O), where

$$\text{C} = \{\text{mapU, scanU, scandU, mealyU, mooreU, zipU, zipUs, zipWithU, unzipU, sourceU, sinkU, initU}\}$$
$$\text{O} = \{\|, \circ, FB_p\}$$

We have also implemented the SMoC and TMoC that are detailed in Mathaikutty [2005].

2.3.2 *Interfacing MoCs in SML-Sys.* It is evident that two different PNs constructed in the same MoC domain may have different timing information. So when connecting them, the relation between their timing must be defined. This relation can be constant and simple or it can vary dynamically. We define interface process constructors that enable connecting two processes within the same MoC when the relation between their timing information is a constant and

simple. These constructors perform a sampling functionality. The constructor *inDup* carries out a sampling-up functionality where the constructor emits r events for each input event (up-rating). Similarily, the constructor *inDdown* carries out a sampling-down functionality where the constructor emits one event for r input events (down-rating).

Furthermore, we define interface process constructors that bridge the processes in different MoC domains by adding or removing the timing information. We define the interface processes that add timing information with the prefix *insert* and those that remove timing information with the prefix *strip*. A strip-based interface constructor removes the timing information that they receive on their input signals. For example to obtain an untimed input from a synchronous process, the absent events (⊔) should be removed, and the other events are passed to the input in the same order as they appear. On the other hand, an insert-based interface constructor injects r number of events into the output. The events inserted are absent events, and r is determined based on the evaluation cycle of the process. The details of the formulation and implementation are provided in [Mathaikutty 2005].

2.4 Metamodeling

A metamodeling framework facilitates the description of a modeling domain by capturing the domain-specific syntax and static semantics into an abstract notation called the metamodel. This metamodel is used to create a *modeling language* that the designer can use to construct domain-specific models by conforming to the *metamodeling rules* governed by the syntax and static semantics of the domain and the framework that enables this is called the modeling framework. *Static semantics* refer to the well-formedness of syntax in the modeled language and are specified as invariant conditions that must hold for any model created using the modeling language. The facility of a metamodeling framework and a metamodel put together is called the *metamodeling capability* as in Figure 2. We identify two distinct roles played in a metamodeling framework. The role of a *metamodeler* who constructs domain-specific metamodels and the role of a *modeler* who instantiates these metamodels for creating domain-specific models.

Consider an audio signal-processing domain. The metamodel specifies the following types of entities: microphones, preamps, power amps, and loudspeakers. Microphones contain one output port, preamps and power amps each contain a single input and output port, and a loudspeaker contains a single input port. The metamodel also specifies the following relationships among the types: a microphone's output port may be connected to the input ports of any number of preamps, a preamp output port may connect to any number of power amp input ports, and a power amp output port may connect to one or more loudspeaker input ports. Such syntactic model construction rules are enforced by the metamodel during modeling and the design editor will not allow any objects or relationships not specified in the metamodel to exist. Therefore, if a designer modeling an audio system tries to connect a microphone's output to the input port of the loudspeaker (in reality this might lead to damaging the

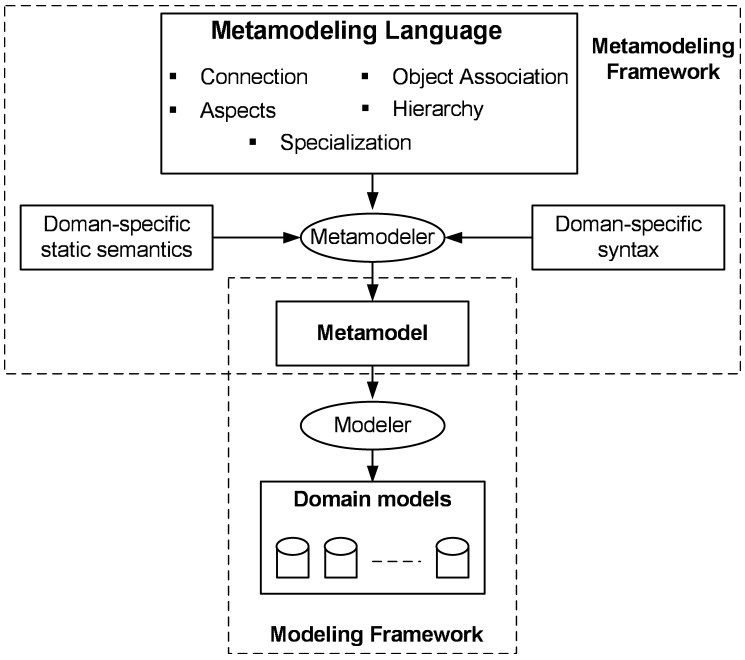


Fig. 2. Metamodeling capability.

loudspeaker), the metamodel will flag a violation and revert the model back to the state it was before the violation occurred.

We extend this notion of metamodeling to a design environment where this capability is used to create and customize MoC-specific metamodels that are used to generate a multi-MoC modeling framework. Design environments such as SystemCH, Ptolemy, Metropolis and EWD allow different degrees of metamodeling, therefore, we measure this characteristic of a design framework based on the ease of customization. An environment that can easily customize the MoC-specific metamodels to enhance or restrict its genericity has a high metamodeling capability. An alternate notion of distinction is based on whether metamodeling is implicitly provided versus it having an explicit capability.

Implicit Metamodeling vs Explicit Metamodeling. A design environment with *Explicit metamodeling* capability must be associated with a metamodeling language, which provides an abstract syntax devoid of implementation details that is customized into MoC-specific metamodels. It expresses a collection of modeling object types, along with the relationships allowed between those object types and the attributes associated with the objects. Furthermore, the framework should enforce the MoC during the modeling activity through metamodeling rules. A design environment has *implicit metamodeling* capability if it does not have a metamodeling language for domain-specific description and makes use of the underlying programming language for the purpose.

Some essential concepts [Nordstrom et al. 1999] of the metamodeling language are shown in Table II.

Table II. Essential Concepts of a Metamodeling Language

Concepts	Description
Connection	Provides rules for connecting objects together and defining interfaces. Used to describe relationships among objects
Aspects	Enables multiple views of a model. Used to allow models to be constructed and viewed from different perspectives.
Hierarchy	Describes the allowed encapsulation and hierarchical behavior of model objects. Used to represent information hiding.
Object Association	Binary and n-ary associations among modeling objects. Used to constrain the types and multiplicity of connections between objects.
Specialization	Describes inheritance rules. Used to indicate object refinement.

When investigating some heterogeneous modeling framework, we find that EWD has explicit metamodeling capability since it is built using GME's metamodeling framework. Ptolemy and SystemCH have implicit metamodeling capability, and the designer is burdened with having to know the underlying programming language (Java, C++) in order to customize the modeling framework. In SystemCH, the task of adding a new MoC requires kernel-level development along with an in-depth understanding of SystemC kernel implementation. Metropolis has a metamodeling language inspired from Java which is nonvisual and based on extensions to abstract classes and class instantiations. $AToM^3$ [Vangheluwe and Lara 2003] is another modeling tool that uses metamodeling and graph-based transformation for model development. They employ metamodeling to define the modeling syntax similar to EWD.

2.5 Generic Modeling Environment

The Generic Modeling Environment (GME) [Ledeczi et al. 2001] is a configurable toolkit that facilitates the easy creation of domain-specific modeling and program synthesis environment. It is proposed as a design environment that is configurable for a wide range of domains and overcomes the high cost associated with developing domain-specific environments such as Matlab/Simulink for signal processing and LabView for instrumentation. The metamodeling framework of GME has a *meta-metamodel* that is a set of generic concepts which are abstract enough such that they are common to most domains. These concepts can then be customized into a new domain such that they support that domain description directly. The customization is accomplished through metamodels specifying the modeling language of the application domain. It contains all the syntactic, semantic and presentation information regarding the domain and defines the family of models that can be created using the resultant modeling framework. These models can then be used to generate the applications or to synthesize input to different COTS analysis tools.

For clarity, we reiterate a few definitions for terms used in the context of GME.

(1) A modeling framework is a modeling language for creating, analyzing, and translating domain-specific models.
(2) A metamodel defines the syntax and static semantics of a particular application-specific modeling framework.

(3) A metamodeling framework is used for creating, validating, and translating metamodels.
(4) A meta-metamodel defines the syntax and semantics of a given metamodeling framework.

The generic concepts describe a system as a graphical, multiaspect attributed entity-relationship (MAER) diagram. Such a MAER diagram is indifferent to the dynamic semantics of the system, which is determined later during the model interpretation process. The generic concepts supported are *hierarchy*, *multiple aspects*, *sets*, *references* and *explicit constraints* which are described using the following constructs.

A *project* (≪Project≫) contains a set of folders that act as containers helping to organize a collection of objects. The elementary objects are called *atoms* (≪Atom≫) and the compound objects that can have parts and inner structure are called *models* (≪Model≫). These objects are instantiated to be a specific kind with a predefined set of attributes. The *kind* describes the role played by these objects and the *attributes* are used to annotate them with domain-specific characterizations. The parts of a *model* are objects of type atom or other models. *Aspects* primarily provide visibility control. Every design has a predefined set of aspects describing different levels of abstraction. The existence of a part of the domain within a particular aspect is determined by the metamodel. The simplest way to express the relationship between two objects in GME is with *connections*. A connection establishes an annotated relation between a set of objects through predefined attributes. *Constraints* in GME are articulated based on the predicate expression language called OCL. They are used to express relationship restrictions as well as rules for the containment hierarchy and the values of the properties (static semantics of the domain).

The authors of Chen et al. [2005] employ GME with GReAT [Ledeczi et al. 2001] and AsmL [Glasser and Karges 1997] to provide structural and behavioral semantics to domain-specific modeling languages (DSML). GReAT provides an infrastructure that allows for model-based transformation with a set of rules governing the manner in which the transformation from one metamodel to another occurs. AsmL is a Microsoft language based on the formal semantics of ASM developed by Gurevich [1995] and Borger and Strk [2003].

The design flow in Chen et al. [2005] contains two metamodels, one for the DSML and the second for AsmL. Input to their system is a model designed following the DSML. The model is then transformed using GReAT into an equivalent model following the AsmL metamodel. Code generation is supported via interpreters whereby executable AsmL code for the model is the output. This can then be simulated using AsmL.

This design flow is in many ways similar to EWD aside from the eloquent use of GReAT to perform the model-based transformations. The formal framework employed in EWD is the functional framework following the generic MoCs and the transformation is done through XML-based parsers as opposed to GReAT and AsmL. It must be noted that at the time of EWD's development, GReAT did not exist. However, transforming the models to SML-Sys or Haskell can also be efficiently done with GReAT.

3. RELATED WORK

3.1 ForSyDe Methodology

ForSyDe [Sander and Jantsch 2004] is another functional programming framework that facilitates synchronous modeling, built on the semantics of SMoC [Jantsch 2003] in Haskell. The ForSyDe design process starts with the development of a high-abstraction formal specification model, which is a perfectly synchronous 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 Sander and Jantsch [2004]. The second phase is the mapping of the implementation model onto a given architecture. This phase is comprised of activities such as partitioning, allocation of resources, and code generation. Synthesizable VHDL and C is generated for HW and SW implementation, respectively. 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 restrict the methodology from automating these refinements.

SML-Sys has a high modeling fidelity since it is a multi-MoC modeling framework based on the generic definition in Jantsch [2003], which is an extension of the ForSyDe methodology. ForSyDe has a low modeling fidelity, since it is based on a single MoC. ForSyDe describes the function-based semantics of MoCs by formulating SMoC in a functional language. It's MoC is based on the synchrony assumption and is best suited for applications amenable to synchrony, which limits it. SML-Sys formulated the untimed, clocked synchronous and timed MoCs as well as the interfacing of these MoCs, and, therefore, is suited for a wide variety of applications.

3.2 Ptolemy II

Ptolemy II supports multiple modeling domains that facilitate designers to model in a truly heterogeneous manner, therefore it is a multi-MoC modeling framework. Every behavior in Ptolemy II is realized by a MoC [Brooks et al. 2005] and a combination of MoCs can be used to describe a system. Ptolemy II's multi-MoC framework follows an actor-oriented modeling approach, which comes with a Java-based graphical user interface (GUI) through which designers can drag-and-drop actors to construct models. Ptolemy II's action-oriented approach [Neuendorffer 2005] has a notion of atomic and composite actors. The atomic actors describe an atomic unit of computation, and the composite actors describe a medium through which more complex computation described by other atomic and composite actors can be hierarchically modeled. In Ptolemy II, *director*s implement the MoC behavior and simulate the model. A model or composite actor that follows a particular director is said to be a part of that MoC's domain.

An attractive quality of Ptolemy II is that it allows for hierarchical heterogeneity [Eker et al. 2003; Brooks et al. 2005] that proposes a better approach

to structure heterogeneous designs. The main idea behind hierarchical heterogeneity is to encapsulate different heterogeneous behaviors within components which can then be hierarchically composed together to complete the description of the design. With this approach, one director is responsible for simulating the local network of actors, whereas another network of actors that contains this component may employ a director following a different MoC. Through the GUI, a designer can explore domain polymorphism whereby a component's director can be replaced with another director, following a different MoC, without having to alter any other actors of the component.

Very much as Ptolemy II's GUI, EWD promotes a component-based graphical interface for describing designs. EWD also realizes its behaviors as MoCs, but the classification is different compared to Ptolemy II's. A timing-based abstraction is used to describe the generic MoCs in EWD as shown in Section 2.2. This classification does not formalize all MoCs available in Ptolemy II such as the CSP and the CT MoCs. Some of the MoCs such as DE and PN-based MoCs are covered by the generic MoCs. An in-depth discussion on the generic MoCs classification is available in Jantsch [2003] and Mathaikutty [2005].

We perceive Ptolemy II as having a higher modeling fidelity due to its support for numerous MoCs and EWD as having a low modeling fidelity due to its support for only the generic MoCs. However, this does not put EWD at a disadvantage because of the genericity of EWD MoCs, thus allowing several of the Ptolemy II MoCs to be easily mapped to a combination of the process constructors of EWD's MoCs. For example, to describe an FSM in Ptolemy II, a designer employs the FSM MoC, whereas in EWD the Mealy process constructor from the untimed MoC may be used to map the same behavior. As for the expressiveness of these two frameworks, in both, the designer must strictly use the MoC syntax for the modeling purpose and avoid using the underlying programming language.

Ptolemy II's support for domain polymorphism requires the actors to be usable in multiple domains. Therefore, their implementation does not imply any MoC-specific execution semantics. Instead, it is the director's responsibility to define the MoC semantics that execute the actors and manage the communication between actors. Ptolemy II checks an actor's compatibility with a domain using interface automata and its type checking system [Eker et al. 2003]. However, there are special cases when actors are not truly polymorphic, such as the *Integrator* and *DifferentialSystem* in the CT domain. In comparison, EWD does not require any sort of interface automata for checking the compatibility of its process constructors and combinators because each constructor and combinator is specifically typed with its respective MoC. This information is included in the metamodel so that violations may be flagged during model construction.

3.3 Metropolis

Metropolis describes an embedded system design as a network of components and their connections through its metamodel (MMM) [Balarin et al. 2003; The Metropolis Project Team 2004]. EWD is an application-independent design environment that is used for building engineering systems through its

meta-metamodel. Therefore, the Metropolis metamodel has more similarities to the meta-metamodel in EWD.

Metamodeling capability. The Metropolis metamodel implemented in Java provides computation, communication, and synchronization primitives as abstract classes. Some of these primitives are process, netlist, media, statemedia, and quantity manager. EWD's meta-metamodel concepts are expressed as UML class diagrams which support techniques such as hierarchy, multiple aspects, sets, references, and explicit constraints. The Metropolis metamodel [The Metropolis Project Team 2004] is independent of any MoC semantics. A design could be directly described using this metamodel, but a set of MoC-specific platforms are provided to facilitate the system description. EWD's notion of a meta-metamodel (GME's meta-metamodel) is also MoC-independent but, through its metamodels, MoC-specific syntax and semantics are expressed, therefore its metamodel is closer to Metropolis's idea of a platform. EWD's meta-metamodel is only for customization and the design descriptions are done through metamodels.

The Metropolis metamodeling capability is provided by inheritance-based platform derivations, whereas EWD facilitates metamodeling through containment and constraint-based hierarchical decomposition. One of the main advantages of EWD's metamodeling capability is its visualization that we believe eases customizability. A point to note is that EWD enforces metamodeling rules (through its metamodel) that the modeler has to strictly follow during modeling. These static semantics [Nordstrom et al. 1999] cannot be enforced using MoC platforms in Metropolis.

Design methodology. The difference between EWD's MoC-specific metamodel and Metropolis MoC-based platform is in how they facilitate the modeling activity. In Metropolis, the user extends the basic process blocks to describe the needed functionality and expresses the communication through a MoC-specific medium. In EWD, the process constructors are instantiated (MoC-specific) and annotated with behavioral code fragments and communication is expressed by instantiating process combinators. Note that process constructors are embedded with MoC semantics, therefore MoC static semantics are enforced through the design editor. However, the behavioral annotations are code fragments specific to a target framework such as SML-Sys or ForSyDe. The extension in Metropolis is based on the MMM, which is independent of a target simulation framework, rendering it more complete than EWD. A Metropolis MoC provides an operational view, whereas EWD's MoC is denotational. The Metropolis infrastructure allows refinements in their platform-based design methodology. Currently EWD does not allow for such a design flow, but the underlying frameworks, SML-Sys and ForSyDe, have refinement libraries that assist semantic-preserving design transformations. Therefore, the multiaspect concept of EWD can be used to customize it for refinements based on these design transformation libraries. The Metropolis compilation begins with the system description that is passed to its frontend compiler to perform syntax and semantic checking and generate an internal representation. This representation is then executed

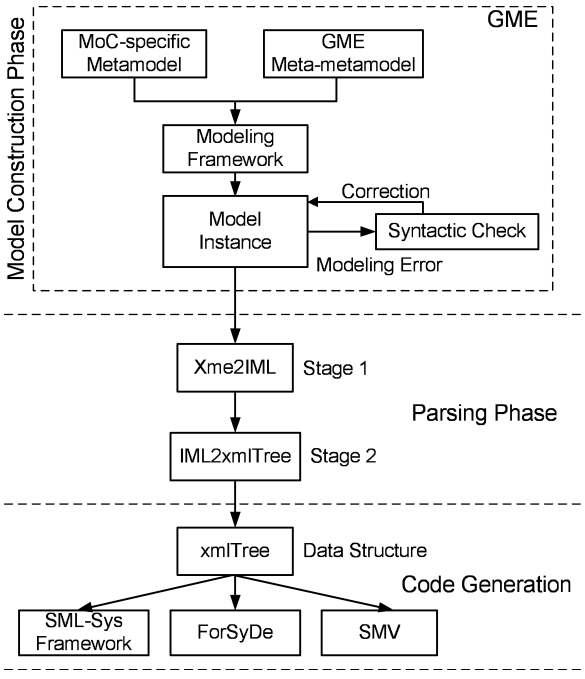


Fig. 3. The design flow.

by specific backend compilers for output that can be given to different simulation, synthesis, or verification tools. The compilation flow in EWD is quite similar to the Metropolis's compilation. The user's model is translated to an internal representation after performing semantic checks. The internal representation is based on XML, which is then translated to the SML-Sys and ForSyDe simulation framework.

4. THE EWD DESIGN FLOW

In **EWD**, the first step in the design flow is the *model construction phase* (MC) followed by the *parsing phase*, which populates our *xmlTree* data structure. Finally, in the *code generation phase*, the *xmlTree* facilitates multitarget simulation as shown Figure 3.

The metamodeler implements the metamodels for the generic MoCs, and the EWD design flow begins with the user being provided with this metamodel for MoC-based modeling as shown in Figure 4.

(a) *MC Phase.* In this phase, there are two different functions performed, one by the user and the other by the design editor. The Modeler Instantiates the metamodel which establishes the framework for the modeling activity and then constructs models constrained by the MoCs. The design editor Performs conformance checks to validate the model against the MoC-specific metamodel. If an error is flagged, the role is transferred back to the user where he/she reworks the model until it conforms to the metamodel

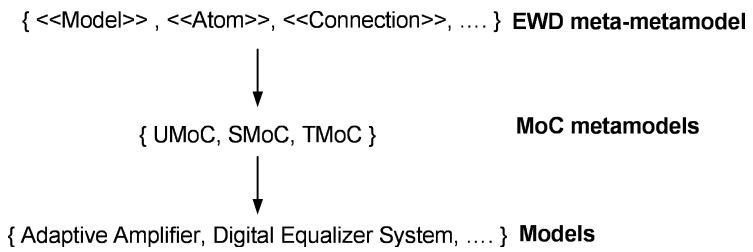


Fig. 4. Usage model.

semantics. Upon completion of this phase, the system converts the model into an XML representation and saves it in a *.xme* extension file.

(b) *Parsing Phase.* We extract the model and metamodel relevant information and represent it in IML. This modeling syntax provides interoperability and implementation independence. The IML-based description of the model is parsed to populate the *xmlTree* data structure.

(c) *Code Generation.* The populated *xmlTree* structure is used to generate SML or Haskell-based executable models depending on the target framework. Henceforth, providing the modeler with a multitarget modeling framework where one can start with a description independent of any programming language and further process it in different design flows. We envision that new design flows can be added with ease in EWD.

5. MODEL CONSTRUCTION PHASE

In this phase, the user instantiates the metamodel for the generic MoC, creating a modeling framework. This framework is composed of different generic computational components and compositional operators. These computational components map to process constructors for our generic MoC, and the compositional operators correspond to process combinators. In our generic MoC, the process constructors are parameterized functions (process templates) in which actual behaviors of the constructor are described through functions passed as arguments. This is mapped to the metamodel through attribute blocks in GME. Each component has a set of attributes attached to it that encapsulates the actual behavior of the component. Furthermore, we target different simulation frameworks, therefore, these behaviors are code fragments written in the language of the target framework. For example, if the simulation is targeting SML-Sys, the different computational components have their behaviors embedded with functions written in SML. The designer uses the drag-and-drop feature to instantiate the components and embed the actual behavior within the attributes, and then uses the appropriate operators to compose them to complete the model. We briefly discuss our metamodel in the following section.

Metamodel for SML-Sys. Our metamodel has the following entities, *ProcessConstructor*, *ProcessNetwork*, *InterfaceConstructor* and *ProcessCombinator*, that are defined using the generic construct ≪Model≫. It also contains an entity *Signal* defined using an ≪Atom≫ construct for describing the inputs and outputs of the model (not shown in Figure 5).

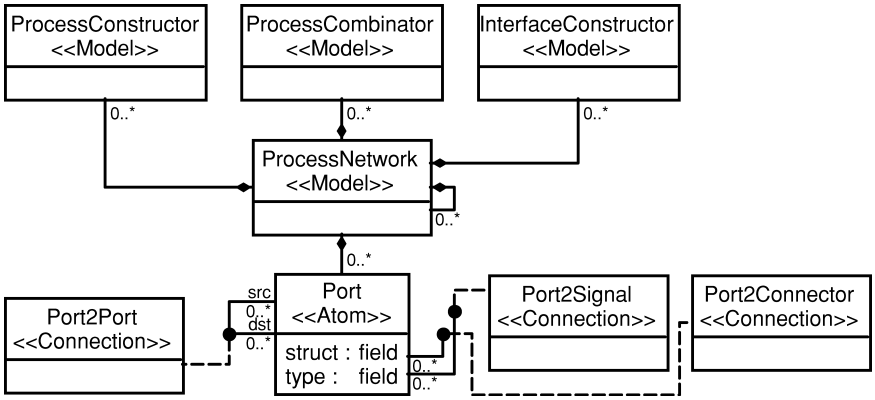


Fig. 5. *ProcessNetwork* entity in the our metamodel.

5.1 *ProcessNetwork* Entity

A *ProcessNetwork* entity as shown in Figure 5 acts as a container for processes that are composed through a set of combinators, and the self-containment relation allows a PN to act as a collection of processes and PNs. Furthermore, it contains interface constructors used for combining processes/PNs across MoCs or with different timing information. A *ProcessNetwork* entity also contains ports (*Port*) defined using the construct ≪Atom≫. A port has two attributes, namely, struct and type, which are used to capture the datatype and structure of the information exchanged. The ports participate in three types of relations, namely, *Port2Port*, *Port2Signal*, and *Port2Connector*. The *Port2Port* ≪Connection≫ is used to compose two PNs, whereas the *Port2Connector* is used to connect a PN to other processes through a process combinator to build a nonelementary PN. The *Port2Signal* is used to connect the PN to the inputs and outputs. Defining these ≪Connection≫s in a PN restrict the modeler from connecting entities by creating incorrect connections. For example, if the modeler wants to compose two processes in a PN without a combinator, the model violates a rule of the metamodel flagging a design time-error.

The metamodeling rules enforced by the PN structure are (i) an elementary PN is made up of two processes and a combinator, (ii) a nonelementary PN is made up of two entities, any of which can be a process or a PN; (iii) processes cannot communicate with each other unless through a process combinator; (iv) if processes that communicate are modeled using different MoC syntax or have different timings, then they should be bridged through interface constructors; and (v) the data communicated through processes/PNs need to be consistent. These metamodeling rules are specified using OCL as shown in Table III & IV and violation of any of the rules in the metamodel will flag a design time-error.

The construct **self** refers to the entity to which a constraint is attached. The function **parts**() collects the children of a parent entity. The first *if-then* segment of the constraint in Table III states that, if the PN contains two processes, then it is an elementary PN and should not contain a PN or an interface constructor

Table III. *Well-Formedness* Constraint for (i) and (ii) of the Metamodeling Rules on a *ProcessNetwork*

```
if(self.parts("ProcessConstructor").size() = 2) then
    (self.parts("ProcessNetwork").size() = 0 && self.parts("InterfaceConstructor").size() = 0)
else if(self.parts("ProcessConstructor").size() = 1) then
    (self.parts("ProcessNetwork").size() = 1 || self.parts("InterfaceConstructor").size() = 1)
else if(self.parts("ProcessConstructor").size() = 0) then
    (
    (self.parts("ProcessNetwork").size() = 2 && self.parts("InterfaceConstructor").size() = 0)
    ||
    (self.parts("ProcessNetwork").size() = 1 && self.parts("InterfaceConstructor").size() = 1)
    )
```

Table IV. One of the *Type* Constraints for (v) of the Metamodeling Rules on a *ProcessNetwork*

```
self.parts("Port")->
    forall(obj1: Port| obj1->attachingConnections()->
        forall(obj2: Port2Port|
            (obj2->connectedFCO("dst").struct = obj1.struct)
            &&
            (obj2->connectedFCO("dst").type = obj1.type)
        )
    )
```

as a child. The metamodel evaluates the constraints as soon as the PN is created and populated. If the constraint evaluates to *false*, then a violation is obtained and the modeler is prevented from proceeding. This constraint is called the *well-formedness* constraint and is specified to allow easy analysis of the structure of the system modeled. However, this results in a highly nested system model, therefore, the user is allowed to disable this constraint and proceed. If the constraint is disabled, then a PN can contain any number of processes, PNs and interfaces which follows from the PN structure are shown in Figure 5.

The construct **forall** is similar to a for loop in C/C++, and it iterates over a set of entities. The function **attachingConnections**() collects all the ≪Connection≫ instances that an entity participates in, and **connectedFCO**() collects the source and destination of a ≪Connection≫ instance. The constraint in Table IV states that ports communicating through a *Port2Port* ≪Connection≫ must have the same values in their struct and type attributes. This enforces that the data communicated is consistent and will not cause type signature errors when translated into SML-Sys/ForSyDe. Similar constraints exist for all ≪Connection≫s that a *ProcessNetwork* entity participates in, and every connection instance is validated at the same time the entities are connected by the modeler.

5.2 *ProcessConstructor* Entity

A *ProcessConstructor* entity is a ≪Model≫ that acts as a place holder for three specialized type of constructors, namely, Single Input Process (SIP), Multiple Input Process (MIP), and Process Initiator (PI) as shown in Figure 6. Note that in the following figures, a triangle means inheritance and a

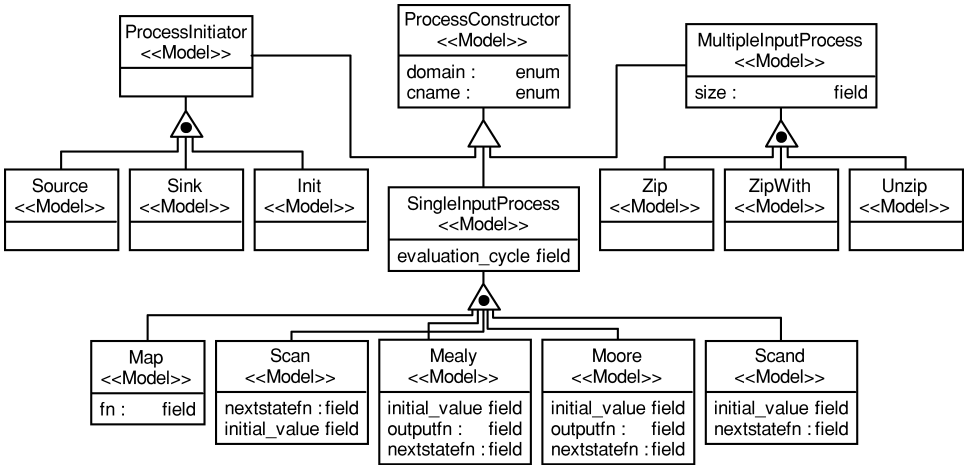


Fig. 6. Categorization of the process constructors.

Table V. Synchrony Constraint on a *SingleInputProcess*

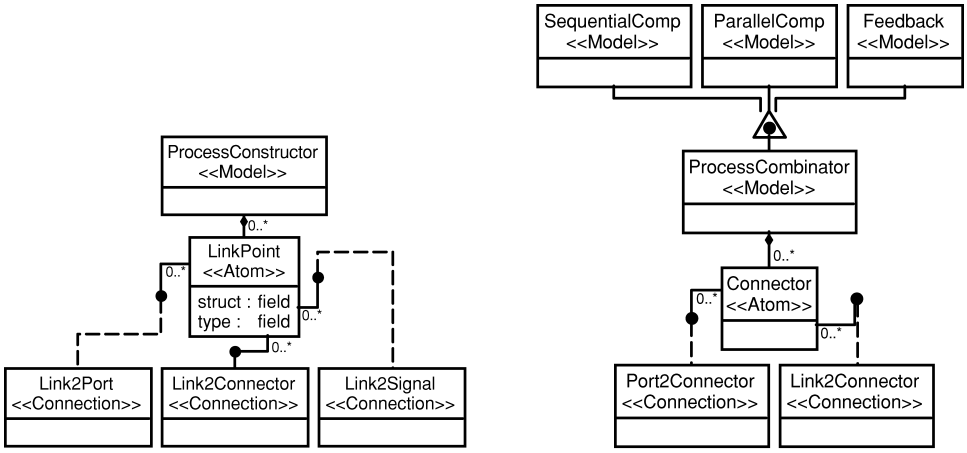
if(**self**.domain = #SMoC) **then self**.evaluation_cycle = 1

triangle with a black dot means implementation inheritance. These constructs are part of GME's meta-metamodel that is used to create our MoC-specific metamodel.

There are five types of SIP constructors, namely Map, Scan, Mealy, Moore, and Scand as shown in Figure 6. Similarly, there are three type of MIP and PI constructors. These different constructors have a set of attributes that are used to capture constructor-specific information to construct the corresponding process. The attribute *domain* and *cname* are common to all the constructors and capture the MoC domain and the constructor name. The *domain* attribute enumerates the following: UMoC, SMoC, and TMoC, one of which is selected by the modeler during the instantiation of the constructor. In the SML-Sys library, there are many implementations for a process constructor, the modeler uses the *cname* attribute to specify the implementation of interest, which is correctly translated to a function call in the appropriate library element. Note that we only show the attributes for the SIP constructors and not the others.

The *evaluation_cycle* attribute captures the events per cycle processed by a constructor. We know that the constructors from the synchronous domain evaluate exactly one event per-cycle and, therefore, there is the *synchrony* constraint shown in Table V that enforces this rule.

Consider the Mealy ≪Model≫ in Figure 6, and the function-based semantics of the mealy-based process in Section 2.3.1. ω_0 is captured by the *evaluation_cycle* attribute, g by *nextstatefn*, f by *outputfn*, and, finally, γ is captured by the *initial_value* attribute. These attributes are extracted in the parsing phase and during translation are used to instantiate and parameterize the correct target process constructor in the respective framework.



(a) *Process Constructor* entity (b) *Process Combinator* entity

Fig. 7. Entities in our metamodel.

Table VI. *MoC mismatch* Constraint on a *ProcessCombinator*

```
self.parts("Connector")->
    forall(obj1,obj2: Connector | obj1 <> obj2 && obj1->attachingConnection()->
        forall(c1: Link2Connector | obj2->attachingConnection()->
            forall(c2: Link2Connector |
              c1->connectedFCO("dst").domain = c2->connectedFCO("src").domain
            )
        )
    )
```

Figure 7(a) shows that the *ProcessConstructor* entity is allowed to have one or more *LinkPoint* entities. A *LinkPoint* ≪Atom≫ represents input and output ports of a process. Therefore, constraints to enforce that SIPs have exactly two *LinkPoint* entities; one each for input and output are part of the metamodel. Similar constraints exist for MIPs which use the *size* attribute (Figure 6) to determine the number of *LinkPoint* entities allowed.

The ≪Connection≫s defined for a *ProcessConstructor* entity are (i) *Link2Signal*, (ii) *Link2Port*, and (iii) *Link2Connector*. The *Link2Signal* allows connecting input/output signals to input/output ports of a process. This ≪Connection≫ includes type constraints for data consistency that enforces equivalence between the *type* and *struct* attributes of a signal and the corresponding port connected to it. The *Link2Port* ≪Connection≫ is used to connect a process to a process network to form a nonelementary PN. The *Link2Connector* ≪Connection≫ is used to connect a process to another process, which is only possible through a combinator captured by the *ProcessCombinator* entity shown in Figure 7(b).

Part of the OCL constraint attached to the *ProcessCombinator* entity that checks for violations from composing processes across MoCs is shown in Table VI. This constraint illustrates the composition of two processes that is checked for a MoC mismatch violation.

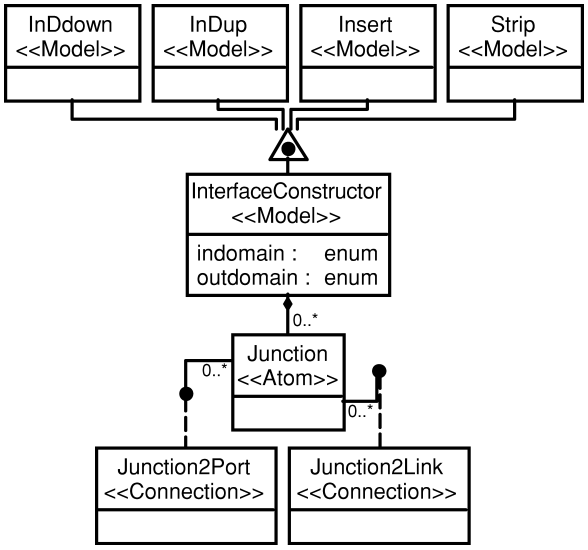


Fig. 8. *InterfaceConstructor* entity in our metamodel.

Table VII. *Interface_chk* Constraint on a *InterfaceConstructor*

```
self.parts("Junction")->
    forall(obj1,obj2: Junction | obj1 <> obj2 && obj1->attachingConnection()->
        forall(d1: Junction2Link | obj2->attachingConnection()->
            forall(d2: Junction2Link |
                (d1->connectedFCO("dst").domain = self.indomain)
                &&
                (d2->connectedFCO("src").domain = self.outdomain)
            )
        )
    )
```

5.3 *InterfaceConstructor* Entity

In Figure 8, we show the *InterfaceConstructor* entity that has four types, namely, *InDup*, *InDdown*, *Strip*, and *Insert*, each of which derives from it. These interface constructors are used to bridge processes/PNs across MoCs or with different timing. Therefore, they should be able to connect with processes through *PointLink*s and to PNs through *Port*s. As a result, an interface is allowed to contain *Junction* entities which acts as connectors and enable the interface to connect with processes through *Junction2Link* and PNs through *Junction2Port*.

The *indomain* attribute captures the MoC name (UMoC, SMoC, TMoC) of the input process and the *outdomain* captures the MoC name of the output process. These attributes are queried to ensure that the correct interface is instantiated to bridge the processes belonging to different MoCs. Part of the *interface_chk* constraint which enforces the check when composing processes across domains. is shown in Table VII.

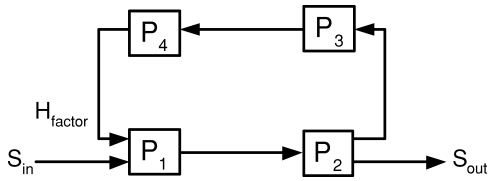


Fig. 9. The adaptive amplifier as a composition of processes.

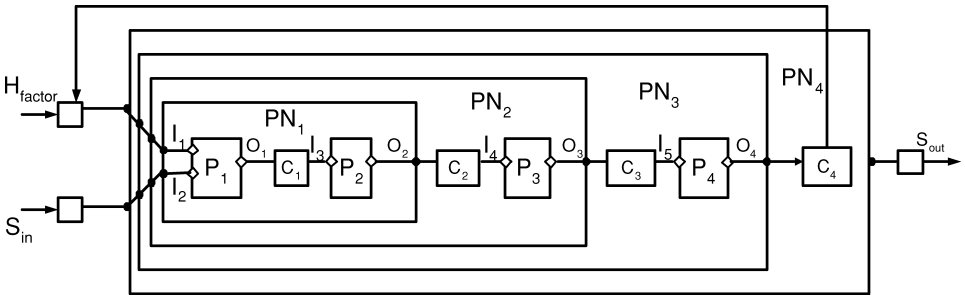


Fig. 10. Model of an adaptive amplifier.

5.4 Modeling an Adaptive Amplifier in UMoC

Figure 9 describes the Adaptive Amplifier (AA) as a composition of four processes. Process P_1 merges the primary input signal with a control signal that contains the amplifying factor. P_2 multiplies the control signal with each element in the primary input signal. P_3 analyzes the amplified signal and adapts the control signal accordingly. P_3 compares the average of the amplified signal to a preferred range, and if it is above a certain threshold, the control signal is lowered. Likewise, if the amplified signal is below the threshold, then the control signal is increased. The adaptive nature is modeled using a feedback loop as shown in Figure 10. The initialization of the feedback loop is performed using the fourth process P_4. This process initializes the output O_4 with the value 10 so that the amplification is within the threshold. All signals carry integer values. We model this example in the untimed domain using our multi-MoC metamodel. We start off by instantiating the UMoC-specific metamodel to create a framework for our modeling activity. We model this as follows: process P_1 is MIP instantiated as a *zip* process, which takes two inputs, S_{in} and H_{factor}. Processes 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, while 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 11. PN_1 is composed using the sequential operator C_2 with P_3 to form PN_2 as shown in Figure 12. PN_2 is composed using the sequential operator C_3 with P_4 to form PN_3 as shown in Figure 13. PN_3 is

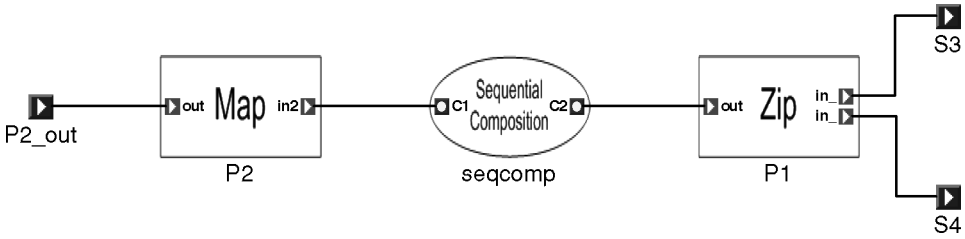


Fig. 11. PN_1 view.

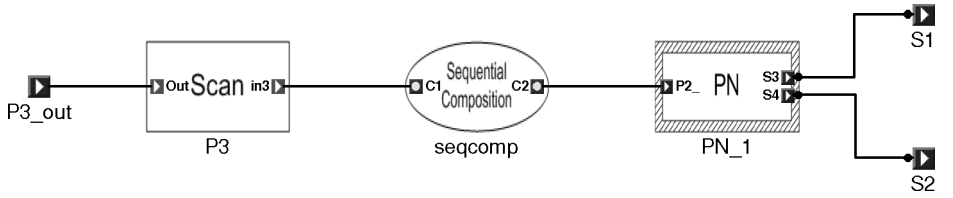


Fig. 12. PN_2 view.

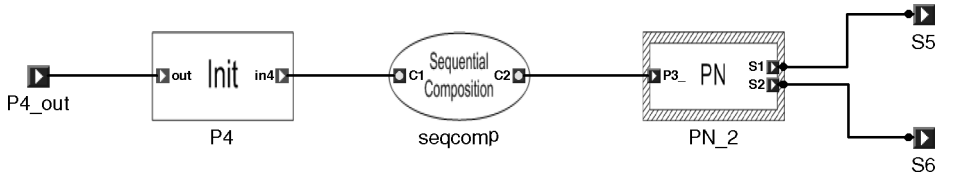
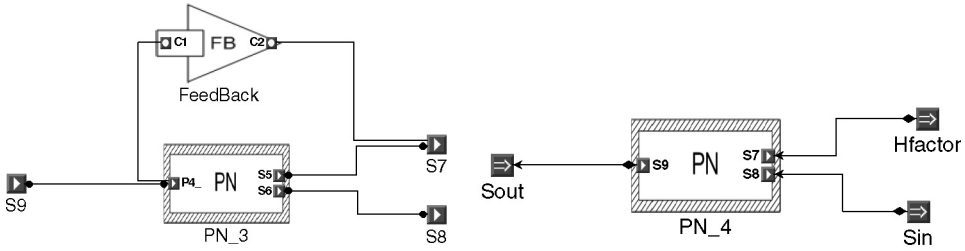


Fig. 13. PN_3 view.



(a) PN_4 View (b) Toplevel View

Fig. 14. Snapshot of PN_4 and top level.

composed with itself in the feedback mode using C_4. Figure 14 illustrates PN_4 and a snapshot of the top level of the AA model.

This is one possible approach to modeling the AA. An alternative is to model the AA as a composition of four processes where the processes are at the same hierarchical level. The pros and cons of the different modeling approaches are discussed in Section 8.

6. PARSING PHASE

The multitarget code generation achieved in this phase is through the XML-based interoperability. In GME, the model and metamodel data are exported

to an XML representation and saved in a *.xme* file which needs to be parsed to extract the model-specific data. Therefore, we parse the *.xme* file using the **Xerces-C++** [Apache XML], which is a validating XML parser built in C++.

The parsing is done in two stages. In the first stage, called *Xme2IML*, we parse the .xme file to extract the relevant information of the system model and remove all the presentation and visualization information which is more than 50% of the file's content. This stage is implemented using the SAX (API for XML) where we build an interface on top of the parser to extract and represent in our IML description. The output of the Xme2IML stage is an IML description of the model, which is further parsed in the second stage, called *IML2xmlTree*, to yield a populated *xmlTree* data structure. This is done using another SAX interface built on top of the parser.

6.1 Interoperable Modeling Langauge (IML)

IML is an XML-based representation for generic MoCs. The model representation in XML extracted from the output of the MC phase is validated against the IML, which is a Document Type Definition (DTD) that defines the legal constructs of the language.

In Section 2.2, we discussed the classification of generic MoCs based on the abstraction of time, which is captured through the attributes of the different legal constructs. The IML constructs are generic enough to represent the model regardless of the underlying MoC. The attributes of the different constructs are of type CDATA[3], and the results of this are that these attributes are skipped during DTD validation. This brings two advantages: (i) the MoC-specific information is embedded in these attributes allowing generic constructs in the IML and (ii) the behavioral aspect of the model that are SML or Haskell code fragments is embedded in these attributes. However, the synchronous domain is a specialization of the untimed domain, therefore, it is easy to mimic synchronous modeling using the untimed modeling constructs. In order to avoid any ambiguity, we introduce an attribute *domain* in the IML constructs which allows correct interpretation during code generation. Furthermore, the IML enforces attribute validity for constructs, therefore, attributes defined with a validity type #REQUIRED need to be specified at all times, otherwise the result is an incomplete translation during the code generation phase.

We briefly describe some of the IML modeling constructs in Table VIII. It has a top-level element called **MODEL** that is used to create the system model using MoC syntax. The MoC and framework information are specified through the *domain* and *framework* attributes shown in line 5. A **MODEL** element constitutes PROCESSes, PNs, INPUTs, and OUTPUTs. A **PN** element is built up from a set of processes that compose their inputs and outputs with a set of combinators shown in line 10. A **PN** can contain one or more instances of itself, thereby allowing hierarchical PNs. A **PROCESS** element shown in line 13 is created through a constructor that consumes a set of input signals and produces a set of output signals. A constructor is basically instantiated by one of the following: *map*, *scan*, *mealy*, *moore*, *scand*, *zip*, *zipWith*, *unzip*, *source*, *sink* or,

[3]Attributes of type CDATA are viewed as non-XML tags.

Table VIII. Some Constructs of the IML DTD

```
1   <!ENTITY % ty_cons "Map } Scan } Mealy } Moore } Zip
2   | UnZip | ZipWith | Scand | Init | Source | Sink">
3
4    <!– MODELING CONSTRUCTS FOR GENERIC MOCs –>
5   <!ELEMENT MODEL (PROCESS | PN | INPUT | OUTPUT)* >
6   <!ATTLIST MODEL name CDATA #REQUIRED
7         domain CDATA #REQUIRED
8         framework CDATA #IMPLIED >
9
10  <!ELEMENT PN (PN | INTERFACE | COMBINATOR | PROCESS | INPUT* | OUTPUT*)+ >
11  <!ATTLIST PN name CDATA #REQUIRED domain CDATA #IMPLIED >
12
13  <!ELEMENT PROCESS (INPUT* | OUTPUT* | CONSTRUCTOR)+ >
14  <!ATTLIST PROCESS name CDATA #REQUIRED domain CDATA #IMPLIED >
15
16  <!ELEMENT CONSTRUCTOR (%ty_cons;)+ >
17  <!ATTLIST CONSTRUCTOR name CDATA #REQUIRED
18              process CDATA #IMPLIED
19              domain CDATA #IMPLIED >
20
21  <!ELEMENT Mealy EMPTY >
22  <!ATTLIST Mealy Evaluation_cycle CDATA #IMPLIED
23        NextStateFn CDATA #REQUIRED
24        OutputFn CDATA #REQUIRED
25        InitialState CDATA #REQUIRED >
```

init as shown in Table II. In our SML-Sys framework, process, templates are higher-order functions that support 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 so that they facilitate interoperability by capturing the behavior of a model, independent of the modeling framework. In line 21 of Table VIII, we illustrate the Mealy-based constructor through the XML element **Mealy** with four attributes that have a one-to-one correspondence to the function-based definition in Section 2.3.1 and the UML diagram in Figure 6.

Similarly, the IML contains constructs such as **COMBINATOR**, **INTERFACE**, **INPUT**, and **OUTPUT**, to capture the process combinators, interface constructors, inputs, and outputs of the model.

6.2 Xme2IML & IML2xmlTree

The first stage of the parsing phase analyzes the output of the MC phase to generate the IML representation. The model created using EWD is exported to an XML-based representation that embeds the model and metamodel-specific information. We extract the model-specific information and initialize the MODEL construct of the IML. We initialize the IML construct PN by extracting information from the *ProcessNetwork* entity. The inputs and outputs are extracted from the *Port* entities within the *ProcessNetwork* entity and used to initialize the IML constructs INPUT and OUTPUT of the PN construct. We extract the constructor-specific information such as name, domain, and its arguments from the *SIP*, *MIP*, or *PI* entities and initialize the PROCESS IML construct.

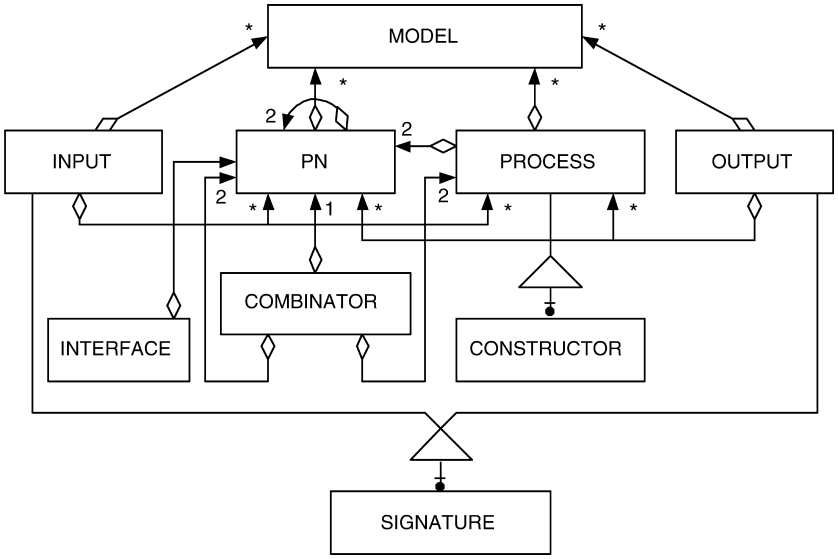


Fig. 15. The class diagram of the *xmlTree* structure.

Similarly the interface-related information is extracted from the *InterfaceConstructor* and used to populate the IML construct INTERFACE. The inputs and outputs to a process are extracted from *LinkPoint* entities. The combinator-related information is extracted from the *ProcessCombinator* entity and used to initialize the IML construct COMBINATOR.

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 class diagram for the *xmlTree* data structure is shown in Figure 15. 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 model in a multitargeted environment. It can further facilitate a reverse design flow where our structure generates the IML description of the model from the target framework.

7. CODE GENERATION PHASE

In the *code generation* phase, the different code generators query the data structure to perform a complete translation into executable models that can be simulated in the targeted framework. The algorithm for the SML-Sys code generators follows:

7.1 SML-Syscode and ForSyDecode

SML-Syscode generator translates the model into SML code that is simulated in the SML-Sys framework. The translation is based on the following algorithm.

SML-Syscode Translation Algorithm

{Given an *xmlTree* G and i,j = 1}
Step 1 Extract G's modeling domain, target framework, PN list and its input/output signals.
Step 2 For $pn \in$ PN list, extract pn_i
  Step 2.1 Extract all input/output signals of pn_i
  Step 2.2 Extract the composition operator of pn_i
  Step 2.3 For process p, s.t. $p \in pn_i$
    Step 2.3.1 Extract all input/output signals of p_i
    Step 2.3.2 Extract constructor type of p_i
    Step 2.3.3 Match constructor type to the respective function in the target framework.
    Step 2.3.4 Extract the arguments of the constructor
  Step 2.4 Extract all domain interfaces
  Step 2.5 For PN pn_j, s.t. $pn_j \in pn_i$, i $\neq$ j Repeat Steps 2.1 to 2.4
    Step 2.5.1 *if* j $\neq$ size of PN list of pn_j then j = j + 1 and Repeat Step 2.4
Step 3 *If* i $\neq$ size of PN list *then* i = i + 1 and Repeat Step 2 and 3
  else Translation Complete

The SML-Syscode generator uses the top-level structure of the *xmlTree* to initialize the skeleton of the model. It further extracts the input and output signals to the model from the top-level. Once the target framework and the modeling domain has been identified, the SML-Syscode encounters a list of PNs in the *xmlTree*. Translation of a PN structure from 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 binding for its components using the SML let construct. The code generator 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. The ForSyDecode generator translates the model into Haskell executables that is targeted for the ForSyDe methodology. The implementation details for the code generation is provided in Mathaikutty [2005].

8. EXPERIENCE WITH EWD AND CASE STUDY

Metamodeling in EWD allows for the creation of multiple MoC-specific metamodels and attaching them to create a library. Adding the MoC-specific interfaces completes the generation of a multi-MoC modeling framework. We have demonstrated this through EWD. Modeling in EWD is as follows. A modeler decomposes the system design into smaller components based on their inherent computational nature. The modeler instantiates the multi-MoC modeling framework and describes the different components using the metamodel that best describe their computation. If the components are described based on different metamodels, then the appropriate interfaces need to be instantiated to integrate and arrive at the complete design. We illustrate the usefulness of EWD by modeling a nontrivial example of a digital equalizer.

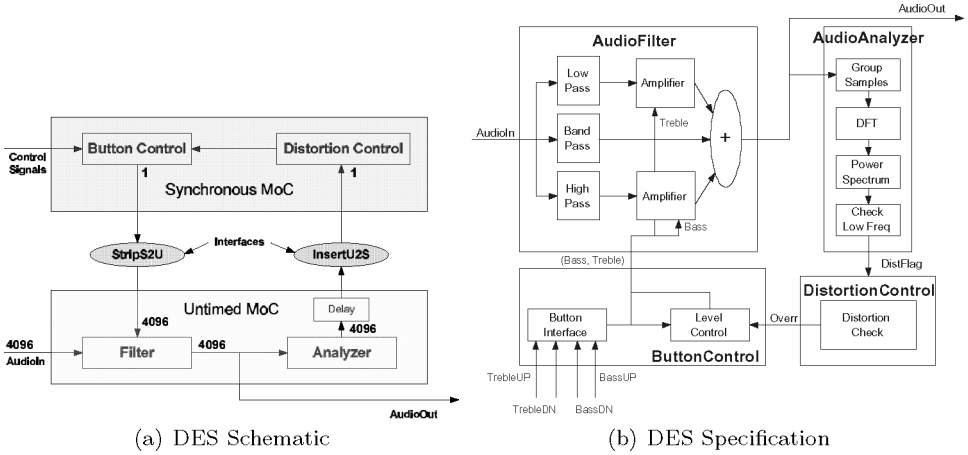


Fig. 16. Digital equalizer system.

8.1 Model of a Multi-MoC Digital Equalizer System (DES)

The task of the equalizer is to modify an audio input signal according to the position of the buttons for the bass and treble levels and to output the modified signal. In addition, the equalizer also monitors the output signal in order to prevent damage to the speakers in case of an extremely high bass level.

The DES consists of a control part and a data part. We model the control part in the Synchronous MoC and a dataflow part in Untimed MoC. Figure 16 [Jantsch 2003] shows the two different parts with the appropriate interfaces to bridge the domains. The specification is decomposed into four components. The *Button Control* subsystem monitors the position of the button inputs and the override signal from the subsystem *Distortion Control* and adjusts the current bass and treble levels. This information is then passed to the subsystem *Audio Filter*, which receives the audio input that filters and amplifies the signal according to the current bass and treble levels. The output signal of the equalizer is analyzed by the *Audio Analyzer* subsystem, which determines whether the bass exceeds a predefined threshold. The result of this analysis is passed to the subsystem *Distortion Control*, which decides if a minor or major violation is encountered and issues the necessary commands to the Button Control subsystem.

The DES has an internal feedback loop and therefore a Delay process has been introduced between the Audio Analyzer and the Distortion Control in order to give the system an initializing value. The interface process between the Analyzer and the Distortion Control is modeled as an Insert-based process that converts the untimed signal to synchronous signal, and the interface between the Button Control and the Filter is modeled as a Strip-based process that converts the synchronous signal to an untimed control signal. While dataflow signals such as AudioIn and AudioOut have a defined value for each event cycle, the control signals BassUP, BassDN, TrebleUP, and TrebleDN are aperiodic and not asserted for most of the time. The interaction of the subsystems is modeled by means of a set of equations where each equation specifies the input and

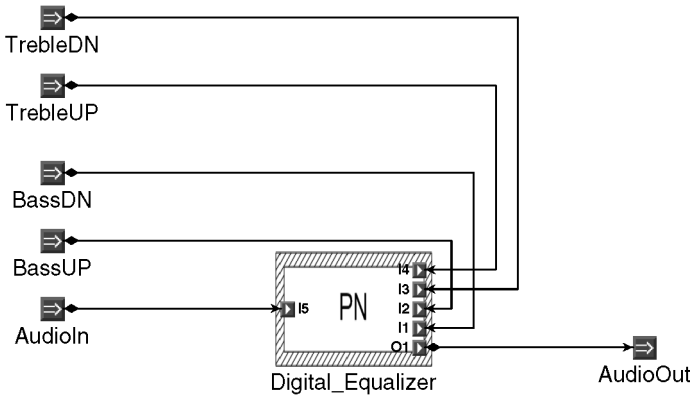


Fig. 17. DES topl evel.

output signals of a subsystem. The formal description follows.

Equalizer (BassDN, BassUP, TrebleDN, TrebleUP, AudioIn) = AudioOut
where,

(Bass, Treble) = ButtonControl (BassDN, BassUP, TrebleDN, TrebleUP, Overr)
AudioOut = AudioFilter (Bass, Treble, AudioIn)
Overr = DistortionControl (DistFlag)
DistFlag = AudioAnalyzer (AudioOut)

The specification of each of the four subsystems are shown in Figure 16 and we systematically illustrate how to model the digital equalizer using EWD[4]. The topl evel of the DES modeled in EWD is shown in Figure 17.

DES internals. The internals of this block are as follows. First, we have the PN *AudioFilter* (AF), sequentially composed with IB_1 where in IB_1 is an encapsulation of the remaining subsystems of the DES as shown in Figure 18. The AF has two inputs and one output. The AudioIn input signal is user-defined and the other input is the output from the Button Control subsystem which comes from IB_1. The output of the AF is given as input to the IB_1 PN through a sequential composition. Furthermore, since IB_1 embeds all the other subcomponents of the system, the inputs to these components are given as input to IB_1. These inputs are BassUP, BassDN, TrebleUP, and TrebleDN, which are given through ports I1, I2, I3, and I4.

IB_1 internals. Figure 19 shows the *AudioAnalyzer* (AA) being sequentially composed with the IB_2 PN, which is the rest of the subsystems of the DES equalizer besides the AF and the AA. The input to the AA is the output of the AF which is also the output of the DES namely AudioOut.

IB_2 internals. A sequential composition of the init-based process and the IB_3 PN in needed to bridge the domains. All the subsystems modeled so far were untimed by nature and since the remaining components of the DES,

[4]The interfaces between the untimed and synchronous subsystems are abstracted in Figure 16.

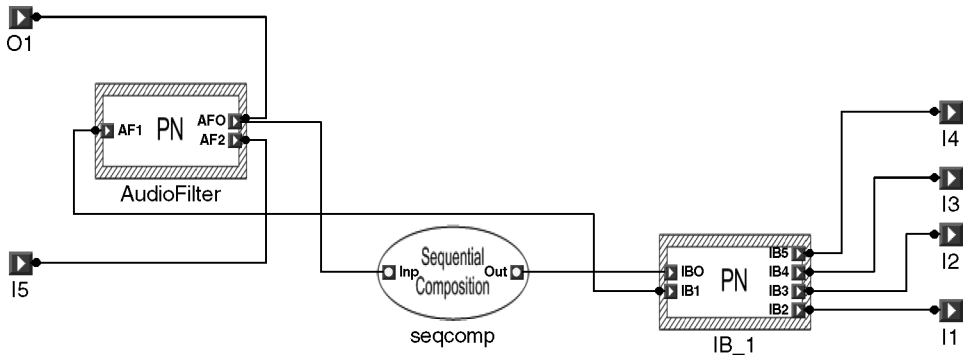


Fig. 18. Audio filter subsystem.

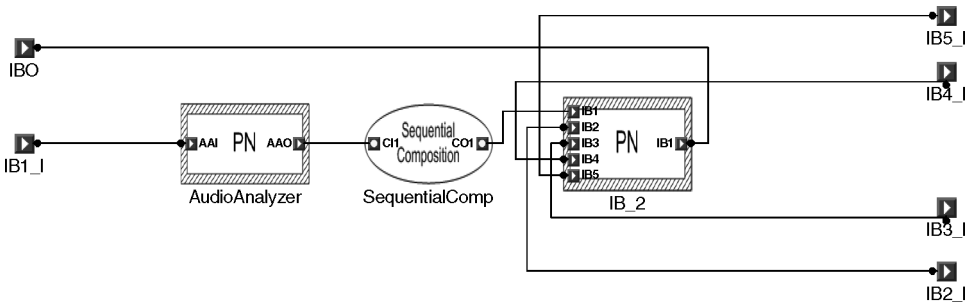


Fig. 19. Audio analyzer subsystem.

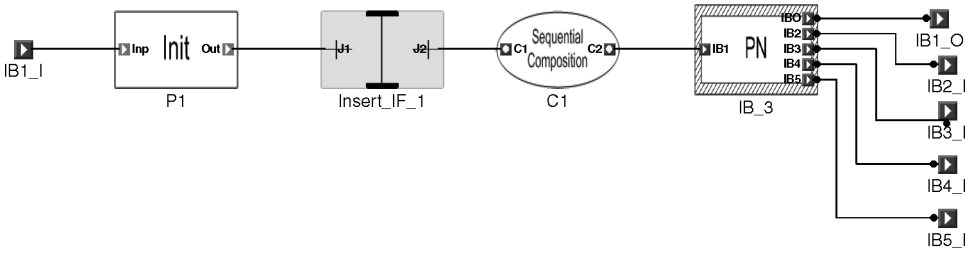


Fig. 20. Bridging MoCs untimed to synchronous.

namely, the Distortion Control and the Button Control, are synchronous, we need to bridge the domain before the output of the untimed domain can be given as input to the components in the synchronous domain. P1 (init-based process), which is the initU-based process, models the delay element shown in Figure 20. The IB_3 PN is basically the remaining synchronous component. The interface to bridge the domains is modeled as an insert-based process (insertU2S), which is the Insert_IF_1 element in Figure 20.

IB_3 & IB_4 internals. The *DistortionControl* (DC) subsystem modeled as a state machine is sequentially composed to the IB_4 PN, which contains the *ButtonControl* (BC) subsystem as shown in Figure 21.

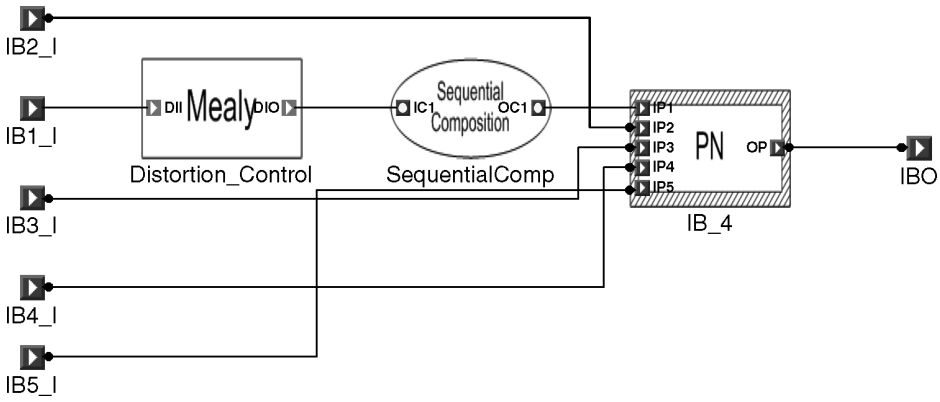


Fig. 21. Distortion control subsystem.

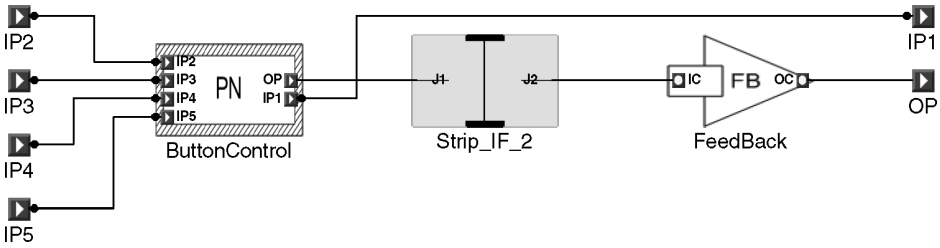


Fig. 22. Button control subsystem and MoC interfacing.

The BC which is composed through a feedback operator to the AF must communicate through interfaces since the BC is modeled within the synchronous domain and the output is given as input to the AF that is untimed. We need the appropriate interface process to bridge these PN, across different domains. The interface process is modeled as a strip-based process (stripS2U), which is the Strip_IF_2 element in Figure 22. We discuss the functionality of the Button Control subsystem and illustrate how we model it using EWD targeting SML-Sys-based code generation.

BC subsystem. The BC works as a user-interface of the DES system. It receives the four inputs BassDN, BassUP, TrebleDN, TrebleUP and the override signal Overr from the DC and calculates the new bass and treble values for the outputs Bass and Treble. The subsystem contains the main processes *ButtonInterface* (BI) and *LevelControl* (LC). The BI process monitors the four input buttons and outputs the value of the pressed button or an absent value if no button is pressed during an evaluation cycle. The LC process keeps track of the current bass and treble levels and adjusts them if either an event from the signal Button or Overr is present. In this case, the process outputs the current levels, otherwise the output value is absent. The process network from Figure 23 is expressed as a set of equations in the formal notation.

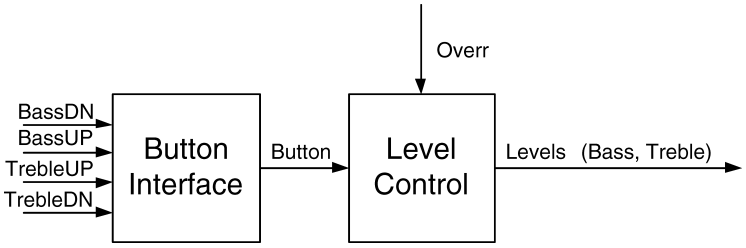


Fig. 23. Schematic of the Button control subsystem.

BC(Overr, BassDN, BassUP, TrebleDN, TrebleUP) = Levels
where,
(Bass, Treble) = unzipS(Levels)
Levels = **LC**(Button, Overr)
Button = **BI**(BassDN, BassUP, TrebleDN, TrebleUP)

Button interface (BI). If two or more buttons are pressed, the conflict is resolved by the priority order of the buttons. If no button is pressed, the output is absent. Since the process is purely combinational and has four inputs, it is modeled by means of a process that is based on the zipWith synchronous constructor as shown.

BI(BassUP, BassDN, TrebleUP, TrebleDN) = zipWithS(condf)(BassUP, BassDN, TrebleUP, TrebleDN)
where,
condf (BassUP, BassDN, TrebleUP, TrebleDN) =
if (bassUp = Active) then BassUP
else if (bassUp = * $\wedge$ bassDn = Active) then BassDN
else if (bassUp = * $\wedge$ bassDn = * $\wedge$ trebleUp = Active) then TrebleUP
else if (bassUp = * $\wedge$ bassDn = * $\wedge$ trebleUp = * $\wedge$ trebleDn = ACTIVE)
then TrebleDN
else

Observe that the use of process constructors simplifies the task for the designer. Since the process constructor (here zipWithS) belongs to the synchronous MoC, the designer has only to formulate the combinational function (f), which will be applied by the process constructor to all values of the incoming signal. The BI is modeled with help of pattern matching.

Level control (LC). The process has a local state expressing the current values for the bass b and treble t. The LC has two modes. In the mode Operating, the bass and treble values are stepwise changed in steps of 0.2. However, there exists a maximum and a minimum value of $+5.0$ and -5.0, respectively. The process enters the mode Locked when the Overr has the value Lock. In this mode, an additional increase of the bass level is prohibited and even decreased by 1.0 in case the signal Overr has the value CutBass. The subsystem returns to the mode Operating on the Overr value Release. The output of the process is an absent extended signal of tuples with the current bass and treble levels.

LC(Button, Overr) = mealyS(nextState, output, (s_0, b_0, t_0))(Button, Overr)
where,

s_0 = Operating
$b_0 = 0$
$t_0 = 0$
nextState((s, b, t), btn, ov) = (s_1, b_1, t_1) with,

$$s_1 = \begin{cases} Locked & \text{if (s = Operating} \wedge \text{ov = Lock)} \vee \text{(s = Locked} \wedge \text{ov} \neq \text{Release)} \\ Operating & \text{if (s = Operating} \wedge \text{ov} \neq \text{Lock)} \vee \text{(s = Locked} \wedge \text{ov = Release)} \end{cases}$$

$$b_1 = \begin{cases} incL(b, Step) & \text{if (s = Operating} \wedge \text{btn = BassUP)} \wedge \text{(ov} \neq \text{CutBass)} \\ decL(b, Step) & \text{if (btn = BassDN)} \wedge \text{(ov} \neq \text{CutBass)} \\ decL(b, cutStep) & \text{if (ov = CutBass)} \end{cases}$$

$$t_1 = \begin{cases} incL(t, Step) & \text{if btn = TrebleUP} \\ decL(t, Step) & \text{if btn = TrebleDN} \end{cases}$$

$$\text{Step} = 0.2$$
$$\text{CutStep} = 1.0$$
$$\text{MaxLevel} = 5.0$$
$$\text{MinLevel} = -5.0$$

$$incL(x, step) = \begin{cases} MaxLevel & \text{if (x + step)} \geq \text{MaxLevel} \\ x + step & \text{otherwise} \end{cases}$$

$$decL(x, step) = \begin{cases} MinLevel & \text{if (x} - \text{step)} \geq \text{MinLevel} \\ x - step & \text{otherwise} \end{cases}$$

$$output((s_1, b_1, t_1), btn, ov) = \begin{cases} \sqcup & \text{if btn} = \sqcup \wedge \text{ov} = \sqcup \\ (b_1, t_1) & \text{otherwise} \end{cases}$$

The process is described by means of the process constructor mealyS, which has a next-state function nextState, an output function output and an initial state as arguments. Since the process constructor belongs to the MoC, the designer has only to formulate the initial state, the next-state and, the output function. The state is divided into a mode (with the initial value s_0 = Operating), a bass value ($b_0 = 0$), and a treble value ($t_0 = 0$). The next-state function can be extracted from the state diagram. The output function selects the bass and treble values from the state in case of a present value of either Button or Overr. Otherwise, the output event has an absent value.

The visual model of the Button Control subsystem and embedding behavior for the Button Interface modeled as a zipWith-based process is illustrated in Figure 24. The other two untimed subsystems, AudioFilter and AudioAnalyzer, and the synchronous DistortionControl subsystem have been modeled in a similar fashion [Mathaikutty 2005].

After the model was complete, conforming to all constraints, we began the parsing phase. During this phase, the model-specific information of DES's four subsystems are extracted and described in IML where the structure of the

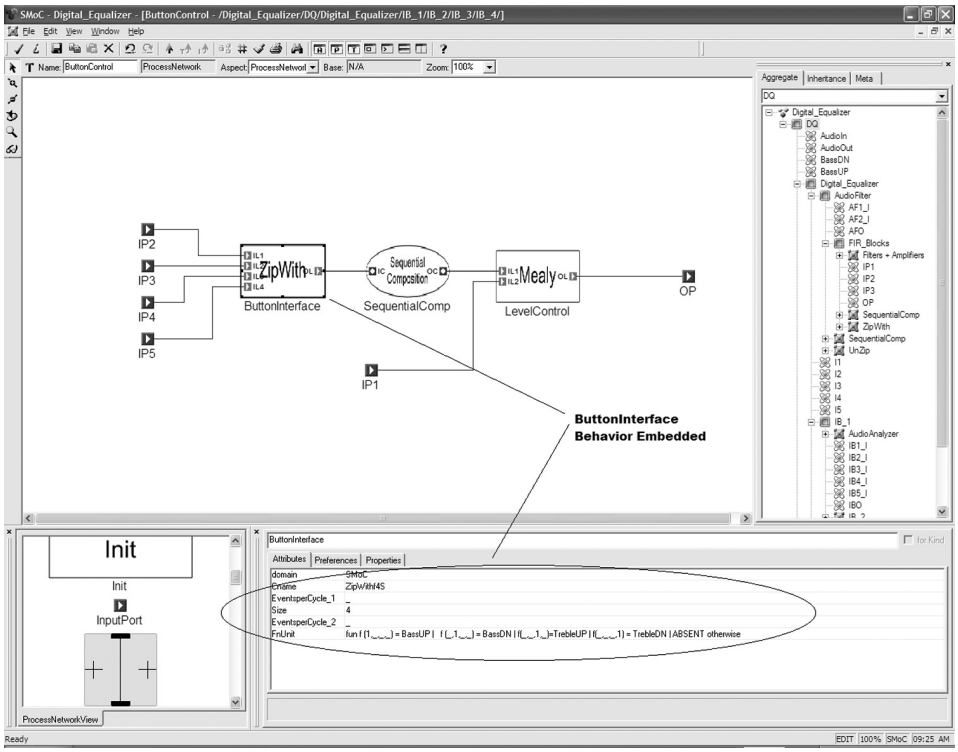


Fig. 24. Button control subsystem modeled in EWD.

model is built by instantiating the corresponding IML constructs and the behavior is embedded into the attributes of these constructs. The IML description of the DES is further parsed to populate the *xmltree* structure which serves as input for the code generation phase. The translation of the DES model into an executable in the SML-Sys framework was achieved through the SML-Syscode generator. The ForDySecode generator cannot be run on the multi-MoC DES model created in EWD because ForSyDe [Sander and Jantsch 2004] is a single MoC-based synchronous framework. In the multi-MoC DES, the AudioFilter and AudioAnalyzer subsystems are modeled as untimed components, therefore, we remodeled these using the synchronous constructs in EWD. The new synchronous DES model was run through the ForDySecode generator to create a simulation model in the ForSyDe framework.

Design experience. From our role as a modeler in EWD, we realized that the usage of our MoC-specific metamodel resulted in highly nested models. This is attributed to the containment relation among the entities. Creating such highly nested models comeswith the tediousness of attaching port-level ≪Atom≫s to each ≪Model≫ entity (*LinkPoint* to *ProcessConstructor*, *Port* to *ProcessNetwork*, etc) that allows for communication across the containment levels. A good amount of the modeling time and effort is spent in correctly constructing these boundaries between the different containment levels, which renders the modeling activity highly prone to errors. The errors were associated

with connecting entities incorrectly, which map to type mismatches in the function signatures generated from the translation to an executable model in the SML-Sys/ForSyDe framework. The OCL constraints on the port entities check that the data communicated is consistent at the datatype level. Many of the modeling errors were caught by these constraints. The uncaught errors tend to mask their error by being consistent at the datatype level. For example, consider ports p_{11} and p_{12} in process P_1 of datatype t and ports p_{21} and p_{22} in P_2 also of datatype t. The modeler connects p_{11} to p_{22} instead of p_{21} making an error, but the identical datatypes mask the error from a type constraint.

The nesting is avoided considerably by creating a flat model where the model is a big PN of many processes composed at the same level. This requires the modeler to disable the well-formedness constraint during model construction. From our case study of creating flat models, we observed that there is a savings in the modeling time, but the translation into an executable was more difficult with new errors that occurred from a flat form representation of the model. Most of the errors were due to overriding of global definitions by local bindings. In a functional language, variable assignment is not allowed since an integral part of the programming paradigm is the absence of the notion of a state. However, a similar idea is provided through name bindings. In SML, the construct *let* is used to create a local environment within a function in which function instances, values, etc, are bound to names that allow reusability. Translation from a nested model resulted in the creation of functions with their own local environments and local bindings. However, the result of translation from a flat model was that the functions were specified at the same level, and this led to the overriding of the global binding by new bindings that were meant to be local. These errors were frequentl, because the function names are overloaded in practice for modeling ease, taking advantage of local bindings.

From our role as a metamodeler in EWD, the following observations were noted. (i) If the MoC metamodel was created to include a lot of modeling details, then the user-interface increases the modeling effort through the tediousness associated with the instantiation and characterization of the entities to create a model that conforms to all constraints. However, one of the advantages of such a metamodel is the large number of constraints that can be specified to enforce a correct construction. (ii) If a generic MoC metamodel was created, then the time necessary for model construction is reduced. However, the number of checks that can be performed is also reduced and the code generators implemented for the translation are more difficult and prone to errors. Therefore, the creation of an MoC metamodel requires attaining the right level of expressiveness where the metamodel is sufficiently rich to allow a wide range of static-MoC checks, and it does not cause the modeling activity to be tedious and time-consuming. Having a metamodeling-driven design environment such as EWD facilitates such an exploration by allowing the metamodeler to create MoC metamodels of varying expressiveness so as to quickly arrive at the desired one with relative ease.

A problem noticed during the implementation of the code generators is the missing notion of sequentiality among the instantiated entities in the visual editor of the modeling framework. When the model is exported into XML, the

entities are collected in the order in which they were instantiated and exported into XML. This forces a sequentiality that is inherent to the exporting tool. This predetermined ordering is often not the correct ordering because the visual framework does not restrict the modeler from throwing the entities onto the editor in any random order and connecting them. Translating such a model would instantiate function definitions in the wrong order which results in illegal function definitions at compile time. Our approach to this problem was to provide constructs in the form of ≪Connection≫s that explicitly specified an ordering and enforced that the modeler use these constructs during modeling. The code generators are implemented to follow the sequential constructs and perform the translation, which is independent of the way the model was exported. However, this approach increases the modeling effort on the part of the user.

Our process constructors are implemented as higher-order functions where a constructor takes secondary functions as parameters. For example, γ, g, f of the Mealy-based constructor are functions specified by the modeler and passed as arguments to the constructor *mealyU*. In our MoC-specific metamodel, these secondary functions are captured through attributes of the *ProcessConstructor* type entities. We have noticed that these attributes are a source for some of the errors during translation, especially since they are not subjected to any datatype-level checking during modeling. Furthermore, they cannot be subjected to any checks because they are specified as plain attributes making it difficult to separate the structure of the function from its actual behavior. The reason for specifying these functions as attributes is that they are code snippets in either SML or Haskell which describes the behavior of the process to be constructed. A solution to this problem is to provide UML constructs to capture the structure separately and allow the behavior to be an attribute. Furthermore, the metamodeler should provide composability checks at the structural level based on how the constructor composes the secondary functions. This would catch most of the errors from the mismatch in the function signature of the secondary functions which are composed by the model at compile time in the SML-Sys/ForSyDe framework.

The constructors provided through our MoCs are sufficient to express varying computations, however it is a common practice to add new constructors which ease the modeling aspect in functional frameworks. These new constructors facilitate specialized computations by extending basic constructors or represent the composition of a set of constructors that is repeatedly employed. This tradition is also possible through EWD and the effort required is minimal due to the ease of customizability. For example, during the development of the DES, one of the new constructors added in SMoC is the $groupS_n$ constructor (Definition 8.1) that was used to model the FIR core in the AudioFilter subsystem.

Definition 8.1. The process group S_n (k) is defined as group $S_n(k) = p$, where

$$p(s_1, \ldots, s_n) = \acute{s} \text{ and } p = mooreS_n\,(f, g, \omega_0) \text{ s.t.}$$

$$f((x_1, \ldots, x_n), \omega_0) = \begin{cases} (x_1, \ldots, x_n) & \text{if } |\omega_0| = k \vee |\omega_0| = 0 \\ \omega_0 \oplus (x_1, \ldots, x_n) & \text{otherwise} \end{cases}$$

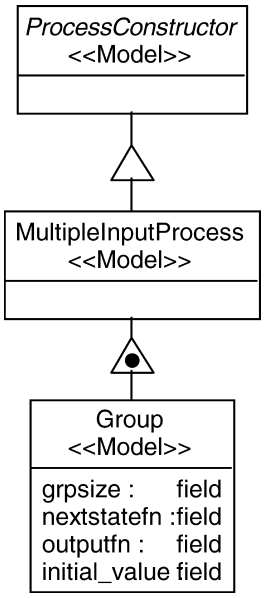


Fig. 25. Addition of *Group* constructor in our metamodel.

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

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

$\oplus$ is the concatenation operator.

The *Group* constructor creates a multiple input Moore-based process which computes the next state based on k events from each of its inputs and the previous state. The metamodel is extended to provide a group-based process creation construct by adding a ≪Model≫ entity that derives from *MultipleInputProcess* and has the same attributes as a *Moore* entity derived from *SingleInputProcess* as shown in Figure 25. The new attribute *grpsize* captures the number of events from each input that needs to be grouped (k). Note that, all the available constraints apply to this new constructor. The metamodeler is also allowed to insert new specialized constraints based on $groupS_n$ constructor's compositional properties. Note that at this point, the modeler plays the role of a metamodeler by customizing the MoC metamodel to suit the modeling needs of the system at hand.

The IML is also extended to capture the new constructor and the new IML construct as shown in Table IX. The extraction and code generation APIs for the Moore-based constructor is reused to extract all attributes of the *Group* entity. The *grpsize* attribute is an integer field in the metamodel similar to the *evaluation_cycle* attribute and therefore is replaced during the usage of the APIs.

Certain observations from our design experience are that the MoC customizability is mostly provided through the addition of new constructors, which are extensions of the basic constructors provided in Table I. However, the addition

Table IX. IML Construct for the Group Constructor

```
1   <!ELEMENT Group EMPTY >
2   <!ATTLIST Group GroupSize CDATA #IMPLIED
2     NextStateFn   CDATA #REQUIRED
4     OutputFn      CDATA #REQUIRED
5     InitialState  CDATA #REQUIRED >
```

of support for a new MoC capability would require defining the metamodel, IML, necessary extraction, and code generation APIs. The creation of the MoC metamodel is easily provided through EWD's meta-metamodel. The IML would be defined in a similar manner using ELEMENTs and ATTLISTs, which is a direct translation from the metamodel. The new APIs would also be very similar to our current APIs since we believe formulating a function-based semantics for an MoC would have similar notions of process constructors, combinators and interfaces.

9. CONCLUSION

Our work in creating the EWD environment has been mainly motivated by the concise denotational semantic of the underlying frameworks such as SML-Sys and ForSyDe. They allow functional design and subsequent formal analysis and transformations towards implementation but require textual programming for system design. The associated tediousness leaves the designers with the need for visualization in such a framework, which not only eases the complete design process but also makes the existing formal semantics a part of the environment through metamodeling.

The metamodeling tool GME motivated us to fulfill this need for visualization on top of a functional modeling framework, and also allowed us to rigorously enforce the metamodel of untimed, synchronous, clocked synchronous, and timed MoCs, together with various static 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. Using GME, we have implemented a metamodel for the generic MoCs with the corresponding interfaces to facilitate a true multi-MoC modeling framework.

Since we already have a generic SML-based environment [Mathaikutty et al. 2004a] for modeling embedded systems, which also has transformation libraries for refinement [Mathaikutty et al. 2004b], and can be subjected to all sorts of static semantic analysis, we have enhanced the visual EWD environment with code generation capabilities. However, to facilitate other functional frameworks such as ForSyDe to use our tool and design flow, we have created an intermediate XML-based language for system description which we call IML. IML parsing and storage in an intermediate data structure and subsequent code generation into SML or Haskell executable models are also described. Finally, EWD requires installation of GME and either the SML multi-MoC libraries or the ForSyDe framework.

Currently, we are developing an imperative programming framework for the function-based semantics of the generic MoCs [Mathaikutty et al. 2005].

The IML language will be used to capture the model representation for interoperability, which is possible since the actual behaviors are captured as code snippets from the modeling language into which the model will be translated for simulation. However, the code generator will be very different from the SML-Syscode or ForSyDecode generator.

APPENDIX

TAGGED SIGNAL MODEL (TSM)

The TSM by Lee and Sangiovanni-Vincentelli [1998] is a denotational framework aimed at analyzing and comparing different MoCs. It is a description that explicitly does not want to provide any execution mechanism for individual processes. Process behavior is exclusively defined by the possible values of their inputs and outputs. Since all processes can have an internal state, the entire history of all inputs and outputs is relevant and called a signal. To give the events in a signal an order of occurrence and to map it onto a global time, a tag system is used such that every event is associated with a tag. If the tags form a partially ordered set with an order relation, events can be related to each other and one event occurs either before, at the same time, or after another event. If the tags of two events are not related with respect to this order relation, it cannot be determined if one event occurs before or after the other. MoCs with a partially ordered tag system are categorized as untimed MoCs and the partial order of events is due to a causality dependence, that is one event triggers the generation of another event, thus it has to occur before the second event.

If the tag system corresponds to a totally ordered set, it can be interpreted as time stamps on a one-dimensional time axis. Then, every event is timewise related to every other event in the system in the sense that the two time tags of two arbitrary events unambiguously determine if one event occurs before, at the same time, or after the other event. If the tags are drawn from a discrete, totally ordered set, for example the set of integers, we have a discrete-time MoC; if the tags are drawn from a continuous, totally ordered set, we have a continuous-time system.

Thus, different structures and properties of the tag system lead to different MoCs (untimed, discrete-time, continuous time). The tag system can further be used to express specific communication properties of processes. For instance, the rendezvous communication mechanism of CSP [Hoare 1978] and CCS [Milner 1989] can be enforced by requiring that there are internal events in two communicating processes that have identical tags when the two processes communicate. This ensures that the two processes are in a well-defined state when they communicate, that is they have a rendezvous.

In summary, TSM provides a complete theoretical framework to express the representation of time in different MoCs; it allows the formulation of constraints on the tag system that captures certain important properties of communication between processes, but it does not deal with execution mechanisms of individual processes because it foremost wants to be a denotational framework that abstracts from concrete process operations.

SML-Sys follows the TSM in that it uses the representation of time as the main criteria to distinguish between MoCs[5]. However, a main motivation for our framework is that it should be of practical use as well as a device for theoretical studies. Consequently, SML-Sys (a) also regulates the execution semantics of processes, (b) is restricted to deterministic processes and process networks, and (c) provides a reasonably practical means to represent global time in timed MoCs.

The execution semantics of processes is defined by process constructors (see Section 2.3.1). It is inspired by the firing mechanism for actors in dataflow process networks [Dennis 1974; Lee and Parks 1995; Lee 1997]. A process is invoked when a sufficient number of events appears on its inputs. We generalized this mechanism also for timed MoCs by allowing absent events to trigger process invocation. This allows a process to be invoked after a specific time period has elapsed, essentially modeling time-outs.

We believe it is beneficial to restrict the framework to deterministic processes and process networks. Formal analysis, verification, and synthesis is simpler and more tractable for deterministic models. For instance, the verification of complex systems is already extremely challenging for fully deterministic models. It is even harder for nondeterministic models because then all nondeterministically possible execution sequences have to be considered even if most of them will not be realized by a concrete implementation or allowed by a realistic system environment. We advocate that nondeterminism should be avoided by choosing the right abstraction level. However, stochastic models can be used to capture unknown delays of communication channels. The pros and cons of deterministic and nondeterministic models are elaborated in more details in Jantsch et al. [2001], but it is worth pointing out that the general metamodeling approach of EWD is not restricted to deterministic models even though the current MoC framework is fully deterministic but includes stochastic processes.

In the TSM, global time is represented by associating a tag with each event. For untimed MoCs, the set of tags is partially ordered, for timed models it is totally ordered. We have opted not to use explicit time tags in our framework for practical reasons. For timed MoCs, the time tag approach does not blend well with our invocation semantics based on firing rules. If a process does not receive any event for some time, it has no means to know how much time has elapsed. Thus, a time-out cannot be modeled where a process is activated if no events are received after a specific time period. However, to be able to model a time-out is essential in a timed MoC. To rectify this problem, a process needs access to global time in some other way, essentially through a global, shared variable. This leads to two other problems. One is the question of how the value of the global variable is distributed to all processes synchronously. The second is that process invocation triggered by a global, shared, variable violates our invocation semantics and leads to nondeterminism.

[5]We treat the untimed model, where causality relations are the only mechanism to order events, as a particular and abstract way to represent time. In the UMoC, time is represented as a partially ordered set. With this convention, we follow Ecker and Hofmeister [1992] and Jantsch et al. [1999]

In summary, since it is nontrivial to maintain a consistent global time in distributed systems [Lamport 1978] and for a deterministic process network model, we opted for embedding the time information into each and every signal such that all signals in the system are synchronous. Absent events are used to maintain synchronicity at time instants when no other event is produced. Note that absent events are a modeling technique that need not be implemented. For the implementation, the synchronicity requirements can be relaxed and a synchronization algorithm for distributed systems, for example as proposed by Lamport Lamport [1978] can be implemented with the help of the occasional emitting of absent events.

To summarize the comparison of the SML-Sys framework with the TSM, we can note the following differences. For untimed MoCs, there is virtually no difference, both models assume a partial order of all events. For synchronous and discrete-time MoCs, our framework embeds the timing information in the structure of the signals, while the TSM associates a global time tag with each event. Both models lead to a global order of time. Our approach matches better with process invocation based on firing rules, is fully deterministic, and does not require implicit access of processes to global time. The TSM easily extends to a continuous-time model, while ours does not. We consider continuous-time MoCs beyond the scope of our current framework, but they may be a subject for a future extension.

REFERENCES

APACHE XML. Xerces-C++ Website. http://xml.apache.org/xerces-c/.

BALARIN, F., WATANABE, Y., HSIEH, H., LAVAGNO, L., PASSERONE, C., AND SANGIOVANNI-VINCENTELLI, A. L. 2003. Metropolis: An integrated electronic system design environment. *In Proceedings of the IEEE Computer Society 36*, 4 (April), 45–52.

BORGER, E. AND STRK, R. 2003. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, Berlin, Germany .

BROOKS, C., LEE, E. A., LIU, X., NEUENDORFFER, S., ZHAO, Y., AND ZHENG, H. 2005. Heterogeneous concurrent modeling and design in java. Memorandum from University of California Berkely Microlab, No. UCB/ERL M05/21.

CHEN, K., SZTIPANOVITS, J., NEEMA, S., EMERSON, M., AND ABDELWAHED, S. 2005. Toward a semantic anchoring infrastructure for domain-specific modeling languages. In *proceedings of the 50 ACM International Conference on Embedded Software (EMSOFT'05)*.

MATHAIKUTTY, D. A. 2005. Functional programming and metamodeling frameworks for system design. Master's thesis, Virginia Tech, Blacksburg, VA.

MATHAIKUTTY, D., PATEL, H., SHUKLA, S., AND JANTSCH, A. 2004a. Correctness preserving design refinements in a functional programming framework for concurrent reactive system design. Tech. rep. 2004-23, FERMAT Lab.

DENNIS, J. B. 1974. First version of a data flow procedure language. Springer Verlag, ed by G. Goos and J. Hartmanis Vol. 19, 362–376.

ECKER, W. AND HOFMEISTER, M. 1992. The design cube—a new model for vhdl designflow representation. In *Proceedings of the European Design Automation Conference*. 752–757.

EKER, J., JANNECK, J. W., LEE, E. A., LIU, J., LIU, X., LUDVIG, J., NEUENDORFFER, S., SACHS, S., AND XIONG, Y. 2003. Taming heterogeneity—the Ptolemy approach. In *Proceedings of the IEEE Special Issue on Modeling and Design of Embedded Software 91*, 1, 127–144.

GLASSER, U. AND KARGES, R. 1997. Abstract state machines semantics of SDL. *J. Found. Comput. Sci.* 3, 12, 1382–1414.

GUREVICH, Y. 1995. *Evolving Algebras 1993: Lipari Guide, Specification and Validation Methods*. Oxford University Press.

HOARE, C. A. R. 1978. Communicating sequential processes. *Comm. ACM 21*, 8, 666–676.

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

JANTSCH, A., KUMAR, S., AND HEMANI, A. 1999. The rugby model: A framework for the study of modelling. In *Proceedings of Design Automation and Test in Europe (DATE)*.

JANTSCH, A., SANDER, I., AND WU, W. 2001. The usage of stochastic processes in embedded system specifications. In *Proceedings of 9th International Symposium on Hardware/Software Codesign*.

LAMPORT, L. 1978. Time, clocks and the ordering of events in a distributed system. *Comm. ACM 21*, 7.

LAVAGNO, L., SANGIOVANNI-VINCENTELLI, A., AND SENTOVICH, E. 1998. Models of computation for embedded system design. *NATO ASI Proceedings on System Synthesis Il*.

LEDECZI, A., MAROTI, M., BAKAY, A., AND KARSAI, G. 2001. The generic modeling environment. Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN.

LEE, E. A. 1997. A denotational semantics for dataflow with firing. Tech. rep. UCB/ERL M97/3, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA.

LEE, E. A. AND PARKS, T. 1995. Dataflow process networks. In *Proceedings of the IEEE 83*, 5, 773–801.

LEE, E. A. AND SANGIOVANNI-VINCENTELLI, A. L. 1998. A framework for comparing models of computation. *IEEE Trans. Comput.-Aid. Design Integ. Circuits Syst. 17*, 1217–1229.

LI, Y. AND LEESER, M. 2000. HML, a novel hardware description language and its translation to VHDL. *IEEE Trans. VLSI Systems 8*, (Feb.). 1.

MATHAIKUTTY, D., PATEL, H., SHUKLA, S., AND JANTSCH, A. 2005. UMoC++: Modeling environment for heterogeneous systems based on generic MoCs. In *Proceedings of the Forum on Specification and Design Languages Conference (FDL)*. Lussanne, Switzerland.

MATHAIKUTTY, D. A., PATEL, H. D., AND SHUKLA, S. K. 2004b. A functional programming framework of heterogeneous model of computations for system design. In *Proceedings of the Forum on Specification and Design Languages (FDL)*. Lille, France.

MILNER, R. 1989. *Communication and Concurrency*. Prentice Hall.

MILNER, R., TOFTE, M., HARPER, R., AND MACQUEEN, D. 1997. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA.

MODELING A DIGITAL EQUALIZER IN THE EWD ENVIRONMENT. 2005. http://fermat.ece.vt.edu/ewd/digital_equalizer.htm.

NORDSTROM, G., SZTIPANOVITS, J., KARSAI, G., AND LEDECZI, A. 1999. Metamodeling—rapid design and evolution of domain-specific modeling environment In *Proceedings of the IEEE*.

PATEL, H. D. AND SHUKLA, S. K. 2005. Towards a heterogeneous simulation kernel for system level models: A systemc kernel for synchronous data flow models. *IEEE Trans. Comput.-Aid. Design 24*.

SANDER, I. AND JANTSCH, A. 2004. System modeling and transformational design refinement in ForSyDe. *IEEE Trans. Comput.-Aid. Design Integr. Circuits Syst. 23*, 1, 17–32.

STEPHEN NEUENDORFFER. 2005. Actor-oriented metaprogramming. Tech. rep. UCB/ERL M05/1.

THE METROPOLIS PROJECT TEAM. 2004. The metropolis metamodel version 0.4. Tech. rep. UCB/ERL M04/38, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA.

THOMPSON, S. 1999. *Haskell—The Craft of Functional Programming*, 2nd ed. Addison-Wesley.

VANGHELUWE, H. AND LARA, J. D. 2003. Computer automated multi-paradigm modelling: Metamodelling and graph transformation. In *Proceedings of Winter Simulation Conference*. 249–258.

Received February 2006; revised November 2006, January 2007; accepted February 2007