Chapter 7

UMoC++: A C++-Based Multi-MoC Modeling Environment

Deepak A. Mathaikutty[1], Hiren D. Patel[1], Sandeep K. Shukla[1], and Axel Jantsch[2]

[1] *Center for Embedded Systems for Critical Applications*
Virginia Tech
Blacksburg, Virginia
USA
[2] *Department of Microelectronics and Information Technology*
Royal Institute of Technology
Stockholm
Sweden

Abstract System-on-chip (SoC) and other complex distributed hardware/software systems contain heterogeneous components that necessitate frameworks capable of expressing heterogeneous models of computation (MoCs) for modeling their functionalities. System-level design languages (SLDLs) that facilitate multi-MoC modeling should have well-defined semantics and should be readily subjected to formal analysis to handle the design complexity. As a result, we follow the multi-MoC paradigm based on timing abstraction and functional parameterizations that have rigorous denotational semantics, which are compliant to functional idioms as shown in functional frameworks such as ForSyDe and SML-Sys. However, functional frameworks are not widely used in the industry due to issues related to efficiency and interoperability with other widely used SLDLs. This imposes a requirement for an imperative language-based implementation of these generic MoCs that offers all the advantages of the underlying formal semantics. In this chapter, we formulate the basis for having generic MoCs in an imperative language and describe the implementation of an untimed modeling framework called UMoC++.

Keywords: model of computation; heterogeneity; untimed; function semantics; functional language.

A. Vachoux (ed.), Applications of Specification and Design Languages for SoCs – Selected papers from FDL 2005, 115–130.
© 2006 *Springer. Printed in the Netherlands.*

1. Introduction

System-level modeling for system-on-chip (SoC) and hardware/(solidus) software codesign has been gaining importance due to the raising complexity of such systems, continual advances in semiconductor technology, and productivity gap in design. Efforts toward mitigating the productivity gap has led to the evolution of several design methodologies of which the one we address are system-level design languages (SLDLs). Examples of recently introduced system-level modeling languages are SpecC, SystemC (OSCI, 2006), and System Verilog (Accellera, 2006). In addition, most system models for SoCs are heterogeneous in nature and encompass multiple models of computation (MoCs) in their different components. As a result, we need a framework that provides a way to express heterogeneous MoCs for modeling SoCs that have well-defined formal semantics and is readily amenable to formal verification.

The multi-MoC paradigm discussed in Jantsch (2003) describes the function-based semantic definition of MoCs that provide a generic classification of computational models by abstracting time. These generic MoCs have well-defined denotational semantics that make them readily subjected to formal analysis and further amenable to functional paradigms. Hence, a functional language (FL) can easily be used to implement them and create a modeling environment. One such modeling framework is ForSyDe (Sander and Jantsch, 2004) built on the semantics of a synchronous computational model (Jantsch, 2003) that facilitates the application of formal methods for transformation and synthesis. However, ForSyDe is more compliant to applications that are synchronous in nature. On the other hand, we need a framework such as SML-Sys (FERMAT, 2006) that provides multiple MoCs for heterogeneous system design. We implemented SML-Sys in a functional language called Standard ML (SML; Milner et al., 1997). Some of the advantages of the SML-Sys framework are: (i) its generic design makes it highly expressible and easily extensible; (ii) precise semantics of the model are derivable because of the rigorous denotational semantics of functional programs; (iii) formal verification of the functionalities of the models can be achieved without having to parse through the C++/Java language specifics; (iv) design transformations are function applications, which provide clean and precise refinement semantics in the framework.

FLs are mainly used in academia for conceptualization and development of theoretical basis for research. It is not common practice to use FLs in the industry for hardware/software design, since the industry prefers imperative languages like C/C++ that have a lower learning curve and provides faster simulation results and requires less memory. The design of imperative languages is based on the von Neumann model, whereas that of FLs are based on mathematical functions. The formulation of generic MoCs in an FL like SML-Sys is inefficient (Okasaki, 1992) due to the implementation cost for single

assignment, call by value and recursion that bring in a lot of implicit garbage collection, potentially huge stack, and lots of copying. Furthermore, designers accustomed to the imperative notion of programming find it difficult to relate and work with FLs due to their abstract expressibility. Most industrial intellectual properties (IPs) are C or C++-based. With the current trend toward IP-based integration, FL-based model development brings in issues of interoperability and reusability. Hence, there is a necessity for an imperative language-based implementation of generic MoCs that takes advantage of features in FL to attain a well-defined formalism and further can be used for cosimulation with other SLDLs to facilitate interoperability and IP-based integration.

2. Related Work

The two most prominent works in classification of MoCs were done in the context of the Ptolemy II project and the ForSyDe project (Sander and Jantsch, 2004). Ptolemy II is built with multiple MoCs, which include various sequential MoCs such as finite state machine (FSM), discrete-time, continuous-time, and MoCs of interacting entities, such as communicating sequential process (CSP), Kahn process network (KPN), etc. The other distinguishing classification of MoCs was done by abstracting time and functional parameterizations. This work can be distinguished from Ptolemy's work as a distinction of the denotational view versus operational view of MoCs (Patel and Shukla, 2004).

ForSyDe is a library-based implementation that provides a computational model for the synchronous domain with interfaces implemented in Haskell. However, since ForSyDe has a single computational model based on the synchrony assumption, it is best suited for applications amenable to synchrony. SML-Sys, when compared with ForSyDe, has a higher modeling fidelity[1] (Patel and Shukla, 2004) since it is a multi-MoC modeling framework based on the generic definition in Jantsch (2003), which is an extension of the ForSyDe methodology.

3. 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. *Processes* receive or consume events and they send or emit events. *Signals* are finite or infinite sequence of events. The activity of processes is divided into *evaluation cycles*. A process partitions its input and output signals into subsequences corresponding to its evaluation cycles. During each evaluation cycle a process consumes exactly one subsequence of each of its input signals. To relate functions on events to

[1] how close it is to the conceptual MoC

processes we introduce process constructors, which are parameterizable structures that instantiate processes. Furthermore, we define process combinators to construct process networks (PNs) through process compositions.

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

Definition 3.1. In Jantsch (2003), an MoC = (C, O) is defined as a 2-tuple where C is a set of process constructors, each of which, when given constructor-specific parameters, instantiates a process. O is a set of process composition operators, which when given processes as arguments instantiates a new process.

MoCs are characterized by the duration of their evaluation cycles. The three generic MoCs defined in Jantsch (2003) are: untimed MoC (UMoC), synchronous MoC, and timed MoC.

Untimed MoC: Processes communicate and synchronize with other processes without the notion of time such that only the order of events are relevant.

Synchronous MoC: The Synchronous MoC divides the timeline into intervals. Every computation within an interval occurs at the same time, but the intervals are totally ordered along the timeline. In synchronous MoCs, the evaluation cycle of processes lasts exactly one time interval. We further categorize synchronous MoCs into:

- **Perfect Synchronous MoC:** This MoC is built on the basis of the *perfect synchrony hypothesis* (Jantsch, 2003), where the output events of a process occur in the same time interval as the corresponding input events.
- **Clocked Synchronous MoC:** This MoC is based on the *clocked synchronous hypothesis* (Jantsch, 2003). It differs from the perfectly synchronous MoC in that every process incurs a delay from an input event to an output event.

Timed MoC: This MoC is a generalization of the synchronous MoC. Timing information is conveyed on the signals by transmitting absent events at regular time intervals.

3.1 Preliminary Notations

We introduce few notations used in defining and distinguishing the generic MoCs. The set of values V represents the data communicated over a signal and the set E constitutes events containing values. A sequence of events constitutes

a signal. Processes are defined as functions on signals p: $S \rightarrow S$ that is a mapping between signal sets. Furthermore, they are allowed to have internal state such that for the same given input signal they react differently at different time instances.

3.2 Generic MoCs Formulation in SML-Sys

In this section, we briefly discuss the implementation of the UMoC for the SML-Sys framework (Mathaikutty et al., 2004). For these generic MoCs, finite signals are implemented as generic lists as shown below:

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

3.3 Untimed Model of Computation

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

Process Constructors. In the UMoC, process constructors are higher order functions that take functions on events as argument and instantiate processes. We implement a set of process constructors that are used to define computational blocks which are either complex processes or process networks. We suffix the name with U to designate it to the UMoC. We discuss the implementation details of a Mealy-based process constructor with respect to finite signals.

Definition 3.2 (Mealy-based process constructor)**.** It resembles a Mealy-based state machine with the addition of a next-state function, an output encoding f that depends on both the input partition and the current state: $mealyU(\gamma, g, f, \omega_0) = p$, where $p(s) = \acute{s}$, with $\Psi(v, s) = \langle a_i \rangle$, where $v(i) = \gamma(\omega_i)$, $g(a_i, \omega_i) = \omega_{i+1}$, $f(\omega_i, a_i) = \acute{a}_i$, and $s, \acute{s}, a_i, \acute{a}_i \in S$, $\omega_i \in E$, $i \in N$.

The list of elementary process constructors implemented for the UMoC is shown in Mathaikutty et al. (2005). The Mealy-based process constructor shown in Listing 7.1 can be extended to handle multiple inputs making it more generic, which results in simplifying the above list.

Process Combinators. We define compositional operators to combine different processes to form complex processes and PNs. These are also implemented as higher-order functions (HoFs). The sequential, parallel, and

Listing 7.1 Mealy-based process constructor in SML-Sys

```
(* mealy-based process constructor for the UMoC *)
fun mealyU (h, g, f, w) = fn (s) => constructor (h, g, f, w, s)

fun constructor (_, _, _, _, []) = [] | constructor (h, g, f, ↵
    w, s) = f (w, head (partition([h w], s))) @ constructor ↵
    (h, g, f, g (w, partition ([h w], s)), drop (s, (h w)))
```

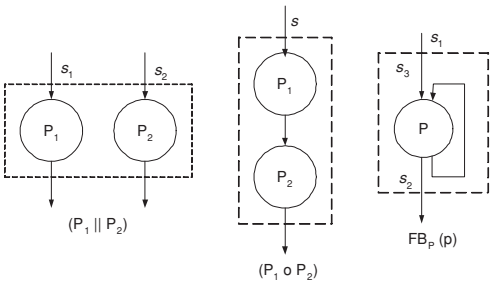


Fig. 7.1 Parallel, sequential, and feedback operators

feedback operators are shown in Figure 7.1. We discuss the implementation of the sequential and the feedback operators with respect to finite signals.

Definition 3.3 (Sequential composition operator)**.** Let p_1 and p_2 be two processes with one input and one output each, and let s ∈ S be a signal. Their sequential composition denoted by $p_1 \times p_2$, is defined as follows: $(p_1 \times p_2)(s) = p_2(p_1(s))$.

```
(* sequential composition operator *)
fun seqcomp (p1,p2) = fn (s) => p2 (p1 (s))
```

Definition 3.4 (Feedback composition operator)**.** Given a process $p : (S \times S) \rightarrow (S \times S)$ with two input signals and two output signals, we define the process $\mathrm{FB}_p(p) : S \rightarrow S$. The behavior of the process $\mathrm{FB}_p(p)$ is defined by the least fixed-point semantics: $\mathrm{FB}_p(p)(s_1) = s_2$, where $p(s_1, s_2) = (s_2, s_3)$.

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

Formalized Definition of UMoC.

Definition 3.5 (UMoC)**.** UMoC is defined as MoC = (C, O), where $C = \{mapU, scanU, scandU, mealyU, mooreU, zipU, zipUs, zipWithU, unzipU, sourceU, sinkU, initU\}$ and $O = \{||, \circ, FB_p\}$

4. Essential Concepts from FL mapped to C++ for our Implementation

The Standard Template Library (STL; Musser and Saini, 2001) is a C++-based generic library of container classes, algorithms, and iterators that are heavily parameterized through templates. Templates allow a generic component to take type `T`, where `T` can be replaced with the actual type. As a result the operations and element manipulations are identical regardless of the type of component, thereby facilitating a way to reuse source code. C++ provides two kinds of templates: class templates and function templates. Function templates are used to write generic functions that can be used with arbitrary types, whereas class templates are usually used as adaptive[2] storage containers. In order to mimic the advantages of FL languages offered by some key concepts like polymorphic types, HoF, and partial applications, we implement similar facilities in C++ using STL skeletons. In the following subsections, these functional concepts and their implementation using C++ are discussed.

4.1 Polymorphic Types

Polymorphism is a type discipline that allows one to write functions that can act on values of multiple types in a uniform way. C++ supports parametric polymorphism by means of templates. A template definition consists of a list of type variables, followed by the definition of a function, a class member function, or a class. Here is an example of a function template to compute the sum of two values:

```
1 template <typename T> T sum(T a, T b) { return a + b; }
```

where `T` is the placeholder for any built-in or user-defined C++ type. The substitution with concrete types is not transparent to the user. The different instances of `sum` are distinguished by the overloading mechanism of C++. For example, if the user passes two integer values to `sum`, the compiler automatically instantiates `int sum(int, int)`. For a user-defined type, the + operator needs to be defined.

[2] reusable and efficient

4.2 Higher-Order Functions

HOFs (Okasaki, 1992) are functions that take functions as arguments and/or return functions as result. Even though the term *higher-order function* is from the FL community, C++ STL also contains many examples of HOFs (e.g., `for_each`, `transform`; Musser and Saini, 2001). HOFs are implemented in C++ using function templates and the ability to overload the function call operator (**`operator()`**). An example of an HOF is shown below where the unary function `f` takes a value of type `x` as argument and applies `f` to `x` once:

```
template <typename OPR, typename ARG> ARG apply(OPR f, ARG x)
{ return f(x); }
```

`OPR` is the placeholder for arbitrary C++ types that support the function call syntax such as pointers to functions and classes that overload the function call operator:

```
template <typename T>
struct Myfunc { T operator() (T c) { return c + c; } };
```

The overloaded **`operator()`** permits objects of type `Myfunc`[3] to be used as if they were ordinary C++ functions. Such objects are called functors or functional objects (Kuchën and Striegnitz, 2002).

4.3 Partial Application

Passing less than n arguments to an n-ary function is called partial application. Partial application (Okasaki, 1992) semantically means binding the first argument of an n-ary function to some fixed value. In C++, support for partial application is limited to binary functions using the `std::bind1st` and `std::bind2nd` constructs. To partially apply a function using these constructs, the user must create a wrapper class for the ordinary C++ function or use the `std::ptr_fun` adaptor that automatically generates an appropriate object. Consider a binary C++ function **`float`** `add(`**`float`** `a,` **`float`** `b)`, which returns `a + b`, using the `bind1st` and `ptr_fun`, we partially apply `add` to `2.0` as shown below:

```
std::bind1st( std::ptr_fun(add), 2.0 );
```

The result of `std::bind1st` is a functional object that behaves like a unary function which takes a single float as argument and returns the value of the argument incremented by `2.0`. For example, instantiating the following will return `12.2`:

```
std::bind1st( std::ptr_fun(add), 2.0 ) (10.2);
```

[3] **`class`** can be used as an equivalent for **`struct`**, except that all members are private by default

The `std::bind2nd` construct can be used similarly to bind the second argument of a binary function. The limitation of partial bind to binary function is overcome with the Boost library (Abrahams and Gurtovoy, 2004) that provides the `boost::bind` construct, which is a generalization of the standard functions `std::bind1st` and `std::bind2nd`. The `bind` construct can handle functions with more than two arguments, and its argument substitution mechanism is more general:

```
1 boost::bind(f, _1, _2, _3)(x, y, z);    // f(x,y,z);
```

C++ offers many features that can be tailored toward the implementation of functional idioms. One problem with the usage of templates to mimic the functional idioms is that of *code bloat* (Musser and Saini, 2001). An alternate approach to this implementation is the use of run-time type information (RTTI), but the problem is the run-time overhead involved.

Many library-based implementations supporting functional programming in C++ are available. The Boost library (Abrahams and Gurtovoy, 2004) provides enhancement to the function object adapters in C++, to support higher-order programming. FC++ (McNamara and Smaragdakis, 2000) is another library-based implementation that allows functional programming in C++ with the reusability benefits of higher-order polymorphic functions.

5. Generic MoCs Formulation in C++

Implementing the function-based formalism of MoCs in C++ brought up the following concerns: C++ allows passing of functions as arguments to other functions in the form of function pointers. However, since function pointers can refer only to existing functions at global or file scope, these function arguments cannot capture local environments.[4] Therefore, we had to model this type of function closure by enclosing the function inside an object such that the local environment or parts are captured as data members of the object. This is possible in C++ because objects in C++ are essentially higher-order records, that is, records that contain not only values but also functions. This sort of abstraction brings in type safety since it avoids the need for *type-casts* or *untyped pointers*.[5] This abstraction is facilitated through C++ classes or class templates. We illustrate how we model our process constructors and process combinators for the UMoC using this abstraction.

[4] the environment captured by a process

[5] an untyped pointer points to any data type

5.1 UMoC++ Framework

In this section, we briefly discuss the implementation of the UMoC in C++. Signals are defined as generic lists, which allows a signal to be of polymorphic type. To facilitate this in C++, we model signals as type `std::vector<T>`, where the placeholder `T` will be replaced by the actual type. Therefore, a signal is a vector of elements of type `T` as shown below:

```
/* Definition of a signal */
template <class T> class signalstruct
{
public:
  signalstruct();
   signalstruct();
  /* Define Accessory functions */
private:
  vector <T> signal;
};
```

We also model the different signal manipulators as function templates that take an input of type `signalstruct<T>` and the other input parameters and generate an output of type `T` or `signalstruct<T>`.

5.2 Process Constructors

We define a set of process constructors that have varied functionalities; some with internal state, some with a single input and output, and some with several inputs and several outputs. The parameters of a process constructor range from an initial state, a next-state function, output encoding function, to partitioning functions for different inputs and outputs. As a result of this varied type of parameters, we use the class template–based abstraction to built objects that hold the local environment for a specific constructor.

The Mealy-based process constructor has been implemented as shown in Listing 7.2. It is a function template that takes two arguments of type `MealyObj` and `signalstruct<SigType>` and returns a value of type `signalstruct<SigType>`. The first argument is an object that captures the environment of Mealy-based process. This class encapsulates an initial state and three member functions, which are as follows: (i) `pfn` determines the number of events handled during an evaluation cycle, from the current state of the process; (ii) `nfn`, when given the current state of the system and the input subsequence, calculates the next state of the system; and (iii) `ofn` is the output-encoding function that produces an output based on the current state and the input subsequence. Therefore, the `mealyU` function template, given an Mealy-based object and an input signal, produces an output and a transition of the system to the next state.

Listing 7.2 Mealy-based process constructor in C++

```
/* Mealy-based Process Constructor */
template <class MealyObj, class SigType>
signalstruct<SigType> mealyU (MealyObj obj, signalstruct ↵
    <SigType> isig)
{
  signalstruct <SigType> osig;
  if(isig.empty() == true)
    return osig;
  else {
    MealyObj tobj;
    osig   = obj.ofn(obj.w, take(isig, obj.pfn(obj.w)));
    tobj.w = obj.nfn(obj.w, take(isig, obj.pfn(obj.w)));
    osig   = append(osig, mealyU(tobj, drop(isig, ↵
        obj.pfn(obj.w))));
    return osig;
  }
}
```

The class template–based abstraction allows us to maintain a uniform structure[6] for the different process constructors, which is essential while defining the process composition operators as explained in the next subsection.

5.3 Process Combinators

Process combinators are operators that define the composition of different process constructors. We describe the implementation of the sequential and feedback combinator. Notice that the class template based–abstraction results in process constructors having similar structure, which facilitates the implementation of generic combinators. This abstraction allows the combinator to take any two processes and an input and define their composition, where the processes passed are objects that encapsulate their functionality.

The implementation of the sequential combinator is shown in Listing 7.3. It is a function template that takes four parameters. The first parameter of type `CombType` describes a type abstraction for the generic processes, whereas the second parameter of type `ProcObj1` and third parameter of type `ProcObj2` are process type objects and the fourth parameter of type `SigType` is an input signal. The first parameter is an encapsulation of two function templates `process1` and `process2`. The output of the instantiation of `process1`, with the object of type `ProcObj1` and the input signal, is given as input to `process2`

[6] w. r. t. number of arguments

Listing 7.3 Sequential composition

```
/* Sequential Composition of Processes */
template <class CombType, class ProcObj1, class ProcObj2, ↵
    class SigType>
SigType seqcomp (CombType comb, ProcObj1 obj1, ProcObj2 obj2, ↵
    SigType isig)
{
  SigType osig;
  osig = comb.process2 (obj2, comb.process1 (obj1, isig));
  return osig;
}
```

along with the object of type `ProcObj2` to return a signal that is the output of the sequential composition.

The implementation of the feedback combinator is shown in Mathaikutty et al. (2005). It is a function template that takes four parameters similar to the sequential combinator, except that the feedback composition is done on a single process and it takes two input signals. The second input signal is the fixed-point signal generated through the fixed-point operator. The fixed-point signal is computed on an event basis, until the fixed-pointing terminates. At each evaluation, the current fixed-point signal depends on the input signal and all the previously generated fixed-point values.

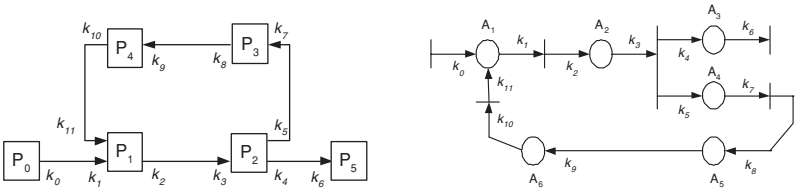
6. Example of Models in our Framework

We model an adaptive amplifier and a power state machine to demonstrate the expressiveness of the framework. The implementations for these models are provided in Mathaikutty et al. (2005). We illustrate the genericness of the UMoC++ framework by describing how to model Petri net, Synchronous data flow (SDF), and FSM.

6.1 Petri Net Style Modeling Using UMoC++

The Petri net style modeling is untimed by nature; therefore, we illustrate how analysis and design techniques can be applied to UMoC++ to allow Petri net modeling. A signature of a process is expressed as a pair of sets (I, O), where I contains the partitioning functions for the input and O contains the partitioning functions for the outputs. For a process network to be mapped to a Petri net, all the processes it is composed of should have constant signatures.

Consider the amplifier process composition shown in Mathaikutty et al. (2005). Let us assume that all processes have a constant signature as shown in Figure 7.2a, then this PN can be converted into a Petri net by representing each process by a transition and each signal by a place as shown in Figure



(a) The Amplifier process with constant partitions

(b) The Petri net for the Amplifier PN

Fig. 7.2 Mapping of the Amplifier PN to a Petri net

7.2b. This mapping is an abstraction of the original process because the data is abstracted into indistinguishable tokens. Consider Figure 7.2, the process signatures (I_i, O_i) for process P_i, $0 \leq i \leq 5$ are as follows:

$$I_0 = \{\} \quad I_1 = \{k_1, k_{11}\} \quad I_2 = \{k_3\} \quad I_3 = \{k_7\} \quad I_4 = \{k_9\} \quad I_5 = \{k_6\}$$
$$O_0 = \{k_0\} \quad O_1 = \{k_2\} \quad O_2 = \{k_4, k_5\} \quad O_3 = \{k_8\} \quad O_4 = \{k_{10}\} \quad O_5 = \{\}$$

with all k_j being constant natural numbers. Each transition in the Petri net represents an input or output of a process, and the weight of a transition is the corresponding constant partitioning function. The restriction on the process signatures can be relaxed to allow processes like state machines that have a rational match at each state[7] such that the ratio can be used to compute the weight of the transitions. Furthermore, these design techniques can be automated since there is a unique mapping from the UMoC++ framework to Petri net.

6.2 Synchronous Data Flow Style Modeling Using UMoC++

The SDF is also untimed and it is a specialization of our generic UMoC. In order to allow SDF style modeling, our UMoC is restricted to where all processes define only constant partitions for all their input and output signals. Therefore, all process signatures are constant. In Section 6.1, we had shown the mapping of PNs to Petri nets. A similar mapping can be used to derive the incidence matrix of the SDF graph, which in turn is used to compute schedules and maximum buffer sizes as explained in Jantsch (2003).

Modeling an FSM in UMoC requires using either the Mealy-based or the Moore-based process constructor. In order to illustrate the FSM capability of the UMoC, we model a fairly complex example of a power state machine as a Moore-based process shown in Mathaikutty et al. (2005).

[7] the ratio of I_q/O_q at state q is a constant

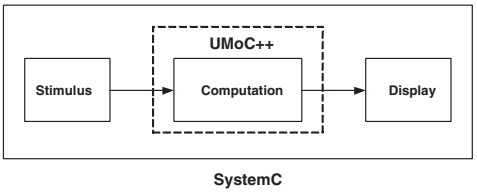


Fig. 7.3 An FIR Filter cosimulated using UMoC++ and SystemC

6.3 Cosimulation With SystemC

In SystemC, a process wraps the functionality that has to be scheduled to run through a discrete-event simulator; therefore, it results in slower simulation. The UMoC++ framework is independent of any simulator, and an immediate advantage is the ability to have an interoperable design approach with C/C++-based SLDLs such as SpecC and SystemC.

The relative ease to cosimulate with the UMoC++ framework is illustrated through an example of a finite impulse response (FIR) filter as shown in Figure 7.3, where the stimulus and display components are simulated using SystemC, and the computational component that convolutes the input and the FIR coefficient is modeled in UMoC++. Since the computation block has to communicate with the stimulus and display block, we built a wrapper for the computation block using SystemC. The computation block is modeled as a sequential composition of a zip-based process that groups the shifted input from the stimulus and the FIR coefficients and a map-based process that convolutes them. Furthermore, this interoperability positions UMoC++ as a framework that facilitates IP-based design integration.

7. Conclusion

We have presented a type-safe framework for modeling in the UMoC using C++, which supports a higher-order functional programming style. As a result, we get the advantages of FL in our imperative-based implementation and also remove the problems associated with efficiency and reusability. The framework is implemented entirely using C++ class templates. Limitation of this framework is that we provide minimal error-checking capability; therefore, the user is required to follow a strict discipline while modeling. Furthermore, during modeling the user is required to use only the generic process constructors and combinators and limit the usage of C++, since one immediate advantage of our framework is its extensibility. In this chapter, we primarily focus on how the foundation for the function-based semantics of generic MoC was built in an imperative language. Furthermore, we implemented the UMoC++ framework, which is a formulation of the UMoC. We illustrate through a set of

examples, the relative ease to model using our UMoC and its generic property that accounts for its expressibility and extensibility. We also briefly describe the mapping of models built in UMoC++ to many untimed variants, such as Petri net, SDF, and provide guidelines for an FSM style modeling. The other generic MoCs such as perfectly synchronous, clocked synchronous, and timed can be implemented using the formulation described in this chapter, which is the context of our future work.

References

Abrahams, D. and Gurtovoy, A. (2004) *C++ Template Metaprogramming: Concepts, Tools and Techniques from Boost and Beyond*. Addison-Wesley, Boston, MA.

Accellera (2006) *SystemVerilog*. Accellera. `http://www.systemverilog.org/`.

FERMAT (2006) *SML-Sys Framework*. FERMAT Group. `http://fermat.ece.vt.edu/SMLFramework`.

Jantsch, A. (2003) *Modeling Embedded Systems and SoC's Concurrency and Time in Models of Computation*. Morgan Kaufmann Publishers, San Francisco, CA.

Kuchën, H. and Striegnitz, J. (2002) Higher-order functions and partial applications for a C++ skeleton library. In: *Proceedings of the 2002 Joint ACMISCOPE Conference on Java Grande*, pp. 122–130.

Mathaikutty, D., Patel, H., and Shukla, S. (2004) A functional programming framework of heterogeneous model of computations for system design. In: *Proceedings of the Forum on Specification and Design Languages (FDL) 2004*, Lille, France. ECSI.

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 (FDL) 2005*. ECSI, Lausanne, Switzerland. pp. 291–302.

McNamara, B. and Smaragdakis, Y. (2000) Functional programming in C++. In: *Proceedings of the International Conference on Functional Programming (ICFP) 2000*.

Milner, R., Tofte, M., Harper, R., and MacQueen, D. (1997) *The Definition of Standard ML—Revised*. MIT Press, Cambridge, MA.

Musser, D. R. and Saini, A. (2001) *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library, 2nd edn*. Addison-Wesley Professional Computing Series. Addison-Wesley, Boston, MA.

Okasaki, C. (1992) *Purely Functional Data Structures*. Cambridge University Press, Cambridge.

OSCI (2006) *Homepage of the SystemC Community*. The Open SystemC Initiative (OSCI). `http://www.systemc.org/`.

Patel, H. D. and Shukla, S. K. (2004) *SystemC Kernel Extensions for Heterogeneous System Modeling: —A Framework for Multi-MoC Modeling & Simulation*. Kluwer Academic Publishers, Boston, MA.

Sander, I. and Jantsch, A. (2004) System modeling and transformational design refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(1):17–32.