Modelling Program-State Machines in SystemC™

Conference Paper · October 2008
DOI: 10.1109/FDL.2008.4641413 · Source: IEEE Xplore

2 authors, including:

Kim Grüttner
OFFIS
89 PUBLICATIONS **270** CITATIONS

SEE PROFILE

Modelling Program-State Machines in SystemC™

Kim Grüttner
OFFIS Research Institute, Oldenburg, Germany
Wolfgang Nebel
CvO University of Oldenburg, Germany

Abstract

The Program-State Machine (PSM) unifies the concepts of hierarchical concurrent finite-state machines, dataflow graphs and imperative programming languages in a single model of computation. It is used as the foundation of the SpecC System Level Design Language. This paper demonstrates the obstacles and proposes an implementation of the PSM model of computation using SystemC. It is shown that this implementation overcomes some fundamental obstacles when using SystemC for System Level Design. Furthermore, we show the applicability of our PSM implementation by porting a JPEG encoder design originally implemented in SpecC. A comparison of model execution time is very promising and shows that our proposed approach is competitive with a native SpecC model execution.

1 Introduction

Over the last years, SystemC™ [10] has gained much interest in academia and industry all over the world. The main reason for this popularity is its ease of application and flexibility. SystemC is not a language but a C++ class library. Thus, the only additional "tools" a designer needs are his favourite text editor and a C++ compiler. In order to perform a simulation, the compiled design is executed on the workstation. No expensive HDL simulator is required anymore. Furthermore, the class library approach makes it easy to use pre-existing C/C++ code in a SystemC model. This is very useful for writing complex testbenches.

One of the main drawbacks of SystemC is that it does not seem to be built as a System Level Design Language (SLDL) right from the beginning. This is reflected by the SystemC core constructs that are rather on a lower level of abstraction. The main building blocks are modules containing plain processes for modelling combinatorial and sequential circuits. Communication is performed through ports that are bound to signals with delta-cycle update semantics. One might get the impression that SystemC has been founded on the experiences of HDLs like VHDL or Verilog that are mainly used for RTL design until today. Over the time, SystemC has gained more high-level constructs that make it possible to use SystemC as a "true" SLDL. In both academia and industry a lot of effort has been spent for the implementation of extensions that bring SystemC closer towards an SLDL. One of the biggest challenges with all these language-oriented extensions is to find a suitable design methodology along with the proposed syntax extensions.

2 Interest & Challenges

One of the most prominent System Design Methodologies [6] is based on the SpecC language combining syntax and a precise semantics [12]. It introduces a minimal set of orthogonal language constructs that can be used for the description and refinement of embedded hardware/software systems. Our work tries to implement the SpecC semantics in SystemC. This approach has been motivated by [4]. It has been shown that in principle the SpecC refinement methodology is applicable to SystemC designs as well. Our work aims to overcome the major obstacles identified therein:

(a) SystemC supports both static and dynamic scheduling. However a static scheduling allows a designer to determine the explicitly modelled execution sequence. This is very helpful during architecture exploration and refinement. SpecC supports only static scheduling using *par*, *pipe* and *fsm* constructs, or default sequential execution. These features are not available in SystemC. Moreover when static sensitivity is used for scheduling in SystemC it affects both the data transfer and the execution sequence (see (d)).

(b) SystemC uses *module* as the structural entity and *process* as the behavioural entity. It does only support hierarchical modelling of processes through explicit process spawning, which is rather tedious and error-prone. Especially the possibilities of unconstrained process spawning are a burden on the designer.

(c) In SystemC variables and events cannot be used to connect ports of different modules. Therefore, they can only be used inside modules or globally. This limits the use of events for scheduling modules and variables for data transfer between modules.

(d) In SystemC signals are used for data transfer between modules. The value of a signal is updated after a delta cycle. This concept makes it difficult to use signals in a specification model (cp. (a)). Therefore, designers should only use dynamic sensitivity for scheduling in specification models using SystemC. This again turns out to be a tedious and error-prone task.

We think that not only SystemC benefits from the SpecC semantics. Also SpecC, which is a superset of C, benefits from SystemC/C++ in the following ways: User-defined data types and operators (encapsulation of data and operations on them) with access protection (public, protected, private), generic and template meta-programming, reuse and specialisation through inheritance & polymorphism (can be applied to both

data and behaviours), access to industry standard C++ libraries like STL and BOOST, and of course reuse of existing C/C++ code. For synthesis of the refined behaviour model state-of-the art SystemC synthesis tools can be used. Moreover, we intend to integrate the proposed extensions into the **O**ldenburg **S**ystem **S**ynthesis **S**ubset [8].

The remainder of this paper is organised as follows. Section 3 introduces the Program-State Machine model of computation. In Section 4 we describe the implementation of PSMs with SystemC. Our implementation is based on some fundamental design principles that impose some limitations, which we discuss at the end of that section. The applicability of our proposed implementation is demonstrated in Section 5 by the implementation of a JPEG encoder design originally written in SpecC. The paper closes with a conclusion.

3 PSM Model of Computation

For the derivation of Program-State Machines (PSM) we take a short look at some of the essential formalisms used in the design of digital systems [6].

A **Finite-State Machine** (FSM) is the most popular model for the description of control systems. An FSM model consists of a set of states, a set of transitions between states, and a set of actions associated with these states or transitions. An extension to eliminate the problem of the state and arc explosion is the introduction of concurrency and hierarchy. This model is called **Hierarchical Concurrent Finite-State Machine** (HCFSM) and is implemented in the widely used StateCharts [9]

A **Dataflow Graph** (DFG) is used for describing computationally intensive systems. Terms that are used to describe computations can be easily represented by a DFG. It consists of nodes that represent operations or functions. Directed arcs between nodes define the execution order.

Finite-State Machines with Datapath (FSMD) combine the features of the FSM and the DFG models. The FSMD is well suited for modelling hardware: Each state transition appears at a single clock cycle and the operations executed in each state can be interpreted as a set of register-transfer operations.

Programming Languages provide a heterogeneous model that supports modelling of data, activity, and control. Programming languages can be classified into two basic paradigms: imperative and declarative. Over the last decade imperative programming languages have become far more accepted than declarative ones. Today, most imperative programming languages support object-oriented features like encapsulation of data and operations, inheritance, and polymorphism. Thus, data is modelled by basic types (e.g. integers and reals), composite types (arrays or structs of basic types) or objects. Activities are modelled by statements, while larger activities can be structured by functions and procedures. Activities on objects are initiated through method calls. Control flow is described by control constructs that specify the order in which activities are to be performed. In imperative languages these are sequential composition (often denoted by a semicolon), branching (`if` and `switch` statements), looping (`while`, `do-while` and `for` statements), and subroutine or method calls.

Merging the FSMD model with the concept of programming languages leads to the so-called **Superstate FSMD** (SFSMD). In this model a superstate does not represent exactly a single clock cycle as in the FSMD model, but any number of clock cycles which depend on the final implementation. Such a superstate can be specified by constructs of programming languages, as mentioned above.

Replacing the FSM model in HCFSMs by a SFSMD model leads to a HCSFSMD, or much shorter **Program-State Machine** (PSM) [5]. A PSM consists of a hierarchy of program-states with each of them specifying a single mode of computation. At each point in time only a subset of program-states is active, and thus perform their computations. A composite program-state can be decomposed into either sequential or concurrent program-substates. The SpecC [12] language implements a PSM model of computation where program-states are called behaviours.

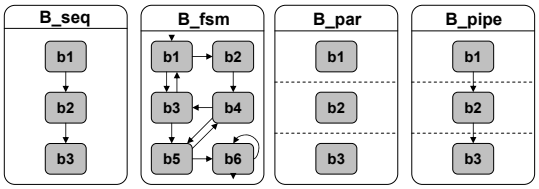


Figure 1: PSM composite behaviours

Figure 1 shows the SpecC composite behaviours. A sequential behaviour describes a purely linear control flow between sub-behaviours. In Figure 1(a) leaf behaviour `b1` is executed until it reaches its completion point, then leaf behaviour `b2` is executed, and so on. `B_seq` is left when `b3` has finished its execution. This kind of composition corresponds to the linear sequential execution as known from any imperative programming language. In the finite-state machine behaviour in Figure 1(b) the execution order of the sub-behaviours depends on the evaluation of the transition arcs. After the initial state `b1` has been entered the successor state can be either `b2` or `b3`, depending on which transitions' guard expression (not shown in this figure) evaluates to true. `B_fsm` is left when `b6` has finished its execution and no transition can be taken.

In a concurrent behaviour, all sub-behaviours become active whenever the parent behaviour is entered. In Figure 1(c) the sub-behaviours `b1`, `b2` and `b3` are executed concurrently when `B_par` is entered. It is left when all concurrent sub-behaviours have finished their executions. This corresponds to the general FORK-JOIN pattern. A combination of concurrent and se-

quential behaviours is the pipeline behaviour. In general a pipeline describes some sort of stream processing in which the same sequence of operations is performed on a stream of data. When `B_pipe` in Figure 1(d) is entered `b1` starts its execution. When it is finished `b1` and `b2` execute in parallel until both of them are finished. It the next step `b1`, `b2` and `b3` execute in parallel until a certain stop condition evaluates to true. Otherwise a pipeline behaviour runs forever.

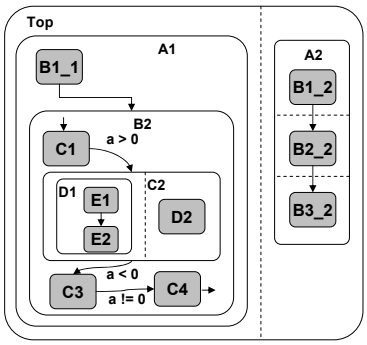


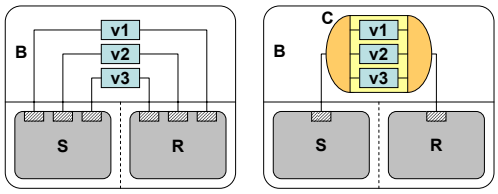
Figure 2: Hierarchical PSM

Figure 2 shows how different composite behaviours can be used to assemble a complex hierarchical PSM.

Until now we have not talked about communication and data-dependent synchronisation. Considering communication in the design of embedded systems along with a strict refinement process towards a physical implementation model the following basic principles can be postulated:

(a) Separation of communication and computation.
(b) Declaration of abstract communication primitives.
(c) Enable custom communication implementation on different levels of abstraction.

SpecC provides channels for the explicit description of communication and thus enables the separation of communication and computation. A channel implements a certain interface that defines which communication primitives are provided. These abstract communication primitives can be used by behaviours that represent the computational parts of the design. Communication is initiated on ports which can be part of behaviour. Only ports whose interface type matches with the interface implemented by a channel can be bound together and therefore establish a communication link. SystemC has adopted this design pattern directly from SpecC.



(a) shared variable (b) virtual channel

Figure 3: Communication in PSMs

Figure 3 shows two different kinds of communication between behaviours. In Figure 3(a) shared variables `v1` to `v3` are bound to ports of behaviour `S` and `R`. Ports can be of direction *in*, *out*, and *inout*. This model of communication corresponds to shared memory communication. In Figure 3(b) a virtual channel that encapsulates the communication using shared variables through a user-defined interface is shown. An example for such a virtual channel is a CSP-like "double handshake channel". This kind of communication corresponds to message passing communication. An extension of a virtual channel is the so-called hierarchical channel. This type of channel is allowed to contain virtual channels and is commonly used for the description of layered protocol stacks used in modern shared buses or network on chips. Besides these data-oriented communication and synchronisation primitives a notification concept based on events can be used as well. Events carry no data and occur only once in time after their notification.

4 PSMs in SystemC

The implementation of PSMs based on SystemC should meet the following design principles:

(a) Compliancy to the IEEE Std-1666™-2005 SystemC standard [10]. Thus, we restrict ourselves to build our extensions only upon the public classes mentioned in the standard. This also restricts us to the SystemC scheduler and process abstraction mechanisms. This imposes some limitations that will be discussed later.
(b) Our proposed extension should fit well into the general rules and coding styles for SystemC code. Especially the syntax of our extensions should fit seamlessly into the standardized SystemC API. While all SystemC language constructs carry the `sc` prefix we have chosen to use the `osss` prefix. Moreover, it should be possible to mix SystemC language constructs with our extensions wherever applicable.
(c) The semantics behind our extension should be equal to the PSM semantics implemented in SpecC. It is desirable to introduce a syntax that is as close as possible to SpecC. This should allow for easy porting designs from SpecC to SystemC using our proposed extensions. To some extent, this requirement is conflicting with the conformance to the SystemC syntax as required above. Considering this, we have tried to use the best from both worlds.

Figure 4 shows a class diagram integrating hierarchical behaviours into SystemC. The `osss_behaviour` class is derived directly from the main structural element of SystemC, the `sc_module` class. Other leaf behaviours, such as pipeline stages (behaviours that can be connected to implement a pipeline) and states (behaviours that can be used to build a finite-state machine) are directly derived from the main behaviour class. Composite behaviours are

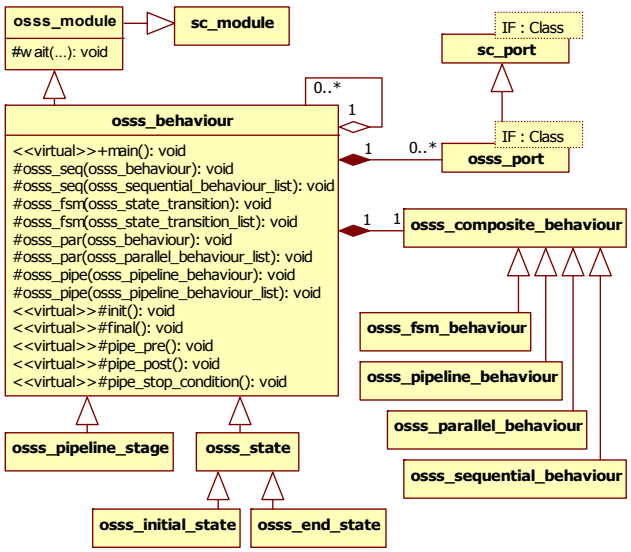


Figure 4: OSSS Behaviour core classes

implemented using the delegation pattern. The `osss_composite_behaviour` class is responsible for the handling of sequential, FSM, parallel, and pipelined composite behaviours. The hierarchy relation itself is inherited from the `sc_module` class since each module is allowed to contain any number of child modules. Communication out of a behaviour is always performed through ports. To better distinguish behaviour-related ports from SystemC built-in ports we have simply derived our `osss_port` class from the SystemC `sc_port` implementation.

Before having a look at a concrete example, we introduce the basic communication and synchronisation primitives. Our implementation follows the "port-interface-channel" concept. Both channels `osss_event` and `osss_shared_variable<Type, Depth>` are derived from the SystemC primitive channel `sc_prim_channel` class. They implement different interface classes to enable read and write accesses. All of these interfaces are derived from the SystemC `sc_interface` class. Finally, port classes derived from `osss_port` that can be bound to different channel interfaces are supplied. In SpecC shared variables can be of any type. Therefore, we have implemented them as a template container class. Shared variables have the advantage over SystemC `sc_signal`s that they do not follow the delta cycle update semantics. When using shared variables for the communication between pipeline stages they need an internal delay representing the number of bypassed pipeline stages. Therefore we have introduced a second template parameter for specifying the depth of the *piped shared variable*. By default shared variables are not piped (i.e. depth = 0). While shared variables are used in specification and early architecture models, signals are used inside channels in a communication model. Therefore, it is important to allow mixing of shared variables and SystemC signals in a model.

```
1  #include <osss_behaviour.h>
```

```
   using namespace ::osss;

   OSSS_BEHAVIOUR(B1_1) {
5    BEHAVIOUR_CTOR(B1_1) {}
     virtual void main() { /* ... */ } };

   OSSS_INITIAL_STATE(C1) {
     INITIAL_STATE_CTOR(C1) {}
10   virtual void main() { /* ... */ } };

   OSSS_STATE(C3) {
     STATE_CTOR(C3) {}
     virtual void main() { /* ... */ } };
15
   OSSS_END_STATE(C4) {
     END_STATE_CTOR(C4) {}
     virtual void main() { /* ... */ } };

20 OSSS_PIPELINE_STAGE(B2_2) {
     osss_in<int>  a_in;
     osss_out<int> a_out;

     PIPELINE_STAGE_CTOR(B2_2) {}
25   virtual void main() { /* ... */ } };
```

Listing 1: Some leaf behaviours of Figure 2

Listing 1 shows some chosen leaf behaviours from the hierarchical behaviour example depicted in Figure 2. We support macros for the definition of behaviour classes and for the default constructors of these classes. This is in compliance to the `SC_MODULE` and `SC_CTOR` macros. Corresponding macros are also provided for other leaf behaviours. The behaviour code itself has to be written into the body of the `main` method. When a behaviour is entered or activated the `init` method is executed once. When a behaviour is left or deactivated the `final` method is executed once.

```
26 OSSS_BEHAVIOUR(D1) {
    protected:
     E1 e1; E2 e2;
    public:
30   BEHAVIOUR_CTOR(D1) { osss_seq(e1, e2); }
   };

   OSSS_STATE(C2) {
    protected:
35   D1 d1; D2 d2;
    public:
     STATE_CTOR(C2) { osss_par(d1 | d2); }
   };

40 OSSS_BEHAVIOUR(B2) {
     osss_in<int> a;
    protected:
     C1 c1; C2 c2; C3 c3; C4 c4;
    public:
45   BEHAVIOUR_CTOR(B2) {
       osss_fsm(
       (c1 >> c2, osss_cond(B2::cond1)) &&
       (c2 >> c3, osss_cond(B2::cond2)) &&
       (c3 >> c4, osss_cond(B2::cond3)) &&
50     (c3 >> c1, osss_cond(B2::cond4)) ); }

     bool cond1() const { return (a > 0); }
     bool cond2() const { return (a < 0); }
     bool cond3() const { return (a != 0); }
55   bool cond4() const { return (a == 0); }
   };

   OSSS_BEHAVIOUR(A1) {
     osss_in<int> in_port;
60  protected:
     B1_1 b1_1; B2 b2;
    public:
     BEHAVIOUR_CTOR(A1) {
       b2(in_port);
65     osss_seq(b1_1, b2); }
   };

   OSSS_BEHAVIOUR(A2) {
     osss_out<int> a_out;
70  protected:
```

```
    B1_2 b1_2; B2_2 b2_2; B3_2 b3_2;
    unsigned int counter;
    osss_shared_variable<int, 1> v0;
    osss_shared_variable<int, 1> v1;

    virtual void init() { counter = 0; }
    virtual void pipe_post() { ++counter; }
    virtual bool pipe_stop_condition() const
    { return (counter == COUNTER_LIMIT); }

  public:
   BEHAVIOUR_CTOR(A2) {
     b1_2(v0);
     b2_2(v0, v1);
     b3_2(v1, a_out);
     osss_pipe(b1_2 >> b2_2 >> b3_2); }
};

OSSS_BEHAVIOUR(Top) {
  protected:
   A1 a1; A2 a2;
   osss_shared_variable<int> var;
  public:
   BEHAVIOUR_CTOR(Top) {
     a1(var);
     a2(var);
     osss_par(a1 | a2); }
};
```

Listing 2: Composite behaviours of Figure 2

Listing 2 contains all composite behaviours from the example in Figure 2. The `Top` behaviour contains the two sub-behaviours `a1` of type `A1` and `a2` of type `A2`. The `osss_par` call performs the parallel composition. Alternatively, the par statement could be inside the body of the `main` method, following the SpecC style. We make use of the dynamic processes of SystemC internally by using the `SC_FORK`, `sc_spawn` and `SC_JOIN` constructs. A shared variable of type `int` is used for communication between `a1` and `a2`. The binding is established by position using the `operator()` in lines 95 & 96.

Behaviour `A2` specifies a pipelined computation using the `osss_pipe` method. The `operator>>()` specifies the execution order of pipeline stages. Usually a pipeline executes in an endless loop. The `pipe_pre` and `pipe_post` (executed before and after each step of the pipeline) in conjunction with the `init` and `pipe_stop_condition` hooks can be used to manipulate the number of pipeline iterations. In `A2` we have specified a simple for-loop that executes the pipeline exactly `COUNTER_LIMIT` times. For the communication between pipeline stages we use pipelined shared variables with a depth of one, since each of them only crosses a singe pipeline stage.

`A1` describes the sequential composition of behaviours `B1_1` and `B2` using the `osss_seq` call. In behaviour `B2` a finite-state machine is specified using the `osss_fsm` method. It takes state transition definitions that are concatenated to a list using the `operator&&()`. The state transition list `(c1 >> c2, osss_cond(B2::cond1))` has the following semantics: A transition from behaviour `c1` to `c2` can be taken when the condition `B2::cond1` evaluates to true. The condition is implemented by a member function of type `bool(void) const`. This syntax might be convenient but is not very close to SpecC. For this reason we have implemented another alternative syntax that looks more like the SpecC equivalent.

During the implementation of PSMs we have encountered some limitations of the chosen approach:

(a) Our approach does not support any of the exception mechanisms (trap and interrupt) available in SpecC. This restriction is due to the underlying non-preemptive SystemC scheduler. Ways to overcome this limitation are described in [2, 1]. However, it requires a major change to the SystemC kernel that might become an extension to the current standard in the future.

(b) The former restriction imposes that we do only support the transition-on-completion (TOC) but no transition-immediate (TI) arcs in the finite-state machine implementation.

(c) SystemC is a C++ class library but no language. It imposes the usage of preprocessor directives (called macros) and restricts us to the syntax of C++ in general. This becomes rather cumbersome when dealing with complicated compiler error messages that rather bare the complexity of compilers than helping the designer. This can be mitigated by implementing various run-time design rule checks which present more useful information.

(d) Complicated debugging since PSMs may involve parallel behaviour that is implemented using a low-level thread model (such as POSIX). But this is a general problem existing for all multithreaded application including SpecC.

5 JPEG Encoder Design

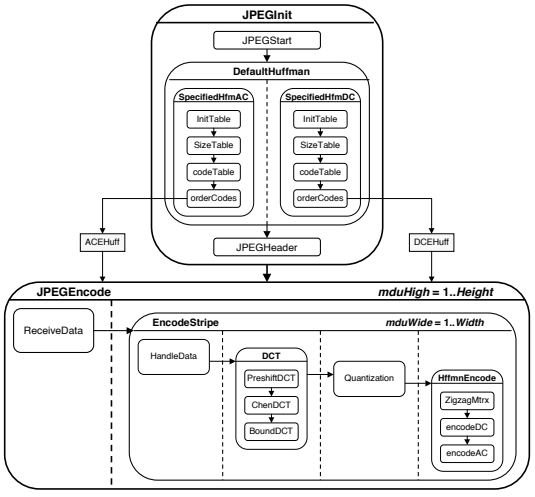


Figure 5: JPEG encoder specification model [7]

For evaluation and comparison a JPEG encoder design [3, 11] has been ported from SpecC to OSSS Behaviour. The original model is described on four levels of abstraction following the SpecC refinement methodology [6]:

(a) A specification (*spec*) model that is untimed (or rather causal-timed) and exploits the parallelism available from the JPEG encoding algorithm, as

model	image dimensions [pixel]			
	116 × 96	256 × 256	461 × 346	512 × 512
OSSS spec[a]	0.088 s	0.455 s	1.103 s	1.762 s
SpecC spec[b]	0.106 s	0.564 s	1.335 s	2.122 s
OSSS arch	0.056 s	0.265 s	0.640 s	0.997 s
SpecC arch	0.109 s	0.593 s	1.423 s	2.226 s
OSSS comm	0.233 s	1.271 s	3.128 s	5.011 s
SpecC comm	1.099 s	6.396 s	15.364 s	25.445 s

[a] with OSCI SystemC 2.2
[b] with SCRC 2.1, both on Intel® Core™2 CPU 6600@2.40GHz

Table 1: JPEG encoder model execution times

shown in Figure 5. This model includes sequential, parallel and pipeline behaviours.

(b) The architecture (*arch*) model after hardware/-software partitioning that is approximately timed. The Discrete Cosine Transform (DCT) is implemented in hardware while all other functionality is implemented in software.

(c) And a communication (*comm*) model where the hardware/software communication is implemented by a cycle and bit accurate model of a bus.

The fourth model, called *implementation model*, is not considered here.

Table 1 shows the different JPEG encoder model execution times measured for input bitmap images of four different dimensions. The results show that our implementation outperforms the SpecC reference implementation. This becomes most apparent when comparing the execution times of the *comm* models. It is important to note that our implementation only executes faster than SpecC when using `SC_THREAD`s with dynamic sensitivity. Using `SC_CTHREAD`s with static sensitivity in combination with `wait()` or `wait(n)` calls reduces the simulation performance tremendously. Therefore, our implementation forbids using that kind of synchronisation. Internally only spawning threads with annotated clock periods are used. Moreover, our implementation provides a `wait` function wrapper and high-level estimated execution time blocks. For more information about implementation details, a proof of concept implementation of our approach can be obtained from `http://www.system-synthesis.org`.

6 Conclusion

In this paper we have discussed the obstacles and proposed an implementation of the PSM model of computation based on SystemC. Our implementation is non-intrusive, which means that it does not make any changes to the standard SystemC kernel. Due to this restriction we were not able to implement the entire functionality of SpecC. In particular the hierarchical exception mechanism can not be supported so far. Nevertheless, our proof of concept implementation is a valuable extension to bare SystemC since it benefits from the SpecC semantics and its methodology. We have introduced a static scheduling using *par*, *pipe*, *fsm* and *seq* constructs that help the designer during architecture exploration and refinement. Moreover, we have made hierarchical processes usable in a well-constrained way. The introduction of shared variables liberates the designer from using bare signals with delta cycle semantics in early specification models. With our extension, events can be encapsulated in the behaviour or module hierarchy since they have become channels instead of global variables. During the implementation we have encountered some general limitations which can only be circumvented with modifications of the SystemC library. Nevertheless, the supported subset is expressive enough for most real world designs. Considering the model execution times of the JPEG encoder design, the proposed approach is absolutely competitive with a native SpecC model execution.

References

[1] P. Arora and R. K. Gupta. Design and Implementation of a Hierarchical Exception Handling Extension to SystemC. In *CASES*, pages 80–84, 2000.

[2] B. Bhattacharya, J. Rose, and S. Swan. Language Extensions to SystemC: Process Control Constructs. In *DAC*, pages 35–38, 2007.

[3] L. Cai, J. Peng, C. Chang, A. Gerstlauer, H. Li, A. Selka, C. S. L. S. S. Zhao, and D. Gajski. Design of a JPEG Encoding System. Technical report, CECS, University of California, Irvine, CA, USA, 1999.

[4] L. Cai, S. Verma, and D. Gajski. Comparision of SpecC and SystemC Languages for System Design. Technical report, CECS, University of California, Irvine, CA, USA, 2003.

[5] D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems*. Prentice Hall, 1994.

[6] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, 2000.

[7] A. Gerstlauer and D. Gajski. System-Level Abstraction Semantics. Technical report, CECS, University of California, Irvine, CA, USA, 2002.

[8] K. Grüttner, C. Grabbe, F. Oppenheimer, and W. Nebel. Object Oriented Design and Synthesis of Communication in Hardware-/Software Systems with OSSS. In *Proceedings of the SASIMI 2007*, October 2007.

[9] D. Harel. StateCharts: A Visual Formalism for Complex Systems. In *Science of Programming*, 1987.

[10] *IEEE Std-1666™-2005 "SystemC™ Language Reference Manual"*.

[11] H. Yin, H. Du, T.-C. Lee, and D. Gajski. Design of a JPEG Encoder using SpecC Methodology. Technical report, CECS, University of California, Irvine, CA, USA, 2000.

[12] J. Zhu, R. Dömer, and D. Gajski. Syntax and Semantics of the SpecC language. In *Proceedings of the SASIMI 1997*, December 1997.