Model Checking Memory-Related Properties of Hardware/Software Co-Designs

Marcel Pockrandt, Paula Herber, Verena Klös, and Sabine Glesner

Technische Universität Berlin
{marcel.pockrandt, paula.herber, verena.kloes, sabine.glesner}@tu-berlin.de

Abstract. Memory safety plays a crucial role in concurrent hardware/-software systems and must be guaranteed under all circumstances. Although there exist some approaches for complete verification that can cope with both hardware and software and their interplay, none of them supports pointers or memory. To overcome this problem, we present a novel approach for model checking memory-related properties of digital HW/SW systems designed in SystemC/TLM. The main idea is to formalize a clean subset of the SystemC memory model using Uppaal timed automata. Then, we embed this formal memory model into our previously proposed automatic transformation from SystemC/TLM to Uppaal timed automata. With that, we can fully automatically verify memory-related properties of a wide range of practical applications. We show the applicability of our approach by verifying memory safety of an industrial design that makes ample use of pointers and call-by-reference.

1 Introduction

Concurrent HW/SW systems are used in many safety critical applications, which imposes high quality requirements. At the same time, the demands on multi-functioning and flexibility are steadily increasing. To meet the high quality standards and to satisfy the rising quantitative demands, complete and automatic verification techniques such as model checking are needed. Existing techniques for HW/SW co-verification do not support pointers or memory. Thus, they cannot be used to verify memory-related properties and they are not applicable to a wide range of practical applications, as many HW/SW co-designs rely heavily on the use of pointers.

In this paper, we present a novel approach for model checking memory-related properties of digital HW/SW systems implemented in SystemC/TLM [15, 20]. SystemC/TLM is a system level design language which is widely used for the design of concurrent HW/SW systems and has gained the status of a de facto standard during the last years. The main idea of our approach is to formalize a clean subset of the SystemC memory model using multiple typed arrays. We incorporate this formal memory model into our previously proposed SystemC to timed automata transformation [13, 14, 21]. With that, we enable the complete and automatic verification of safety and timing properties of SystemC/TLM designs, including memory safety, using the Uppaal model checker. Our approach can handle all important elements of the SystemC/TLM language, including port and socket communication, dynamic sensitivity and timing. Thus, we can cope with both hardware and software and their interplay. We require our approach for model checking of memory-related properties of SystemC/TLM designs to fulfill the following criteria:

1. The SystemC memory model subset must be clearly defined.
2. The automatic transformation from SystemC/TLM to Uppaal must cover the most important memory related constructs, at least the use of pointer variables and call-by-reference.
3. The resulting representation should produce as little overhead as possible on verification time and memory consumption for the Uppaal model checker.
4. To ease debugging, the automatically generated Uppaal model should be easy to read and understand.

Note that our main goal is to make the theoretical results from formal memory modeling applicable for practical applications, and in particular, to transfer the results from the verification of C programs (with pointers) to the verification of HW/SW co-designs written in SystemC. We do not aim at supporting the full C memory model, including inter-type aliasing and frame problems. Instead, we focus on a small, clean subset of the SystemC memory model that is sufficient for most practical examples and can be verified fully automatically.

The rest of this paper is structured as follows: In section 2, we briefly introduce the preliminaries. In section 3, we summarize related work. In section 4, we present our approach for the formalization of the SystemC memory model with Uppaal timed automata. Then, we show how we incorporated the memory model into our previously proposed automatic transformation from SystemC/TLM to Uppaal. In Section 5, we describe the verification of memory safety with our approach. Finally, we present the results of this transformation for our case study in Section 6 and conclude in Section 7.

2 Preliminaries

In this section, we briefly introduce the preliminaries that are necessary to understand the remainder of the paper. First, we give an overview over the system level design language SystemC/TLM and Uppaal timed automata (UTA). Then we give a brief introduction into our transformation from SystemC to timed automata.

2.1 SystemC/TLM

SystemC [15] is a system level design language and a framework for HW/SW cosimulation. It allows modeling and executing of hardware and software on various levels of abstraction. It is implemented as a C++class library, which provides the language elements for the description of hardware and software, and an event-driven simulation kernel. A SystemC design is a set of communicating processes, triggered by events and interacting through channels. Modules and channels represent structural information. SystemC also introduces an integer-valued time model with arbitrary time resolution. The execution of the design is controlled by the SystemC scheduler. It controls the simulation time, the execution of processes, handles event notifications and updates primitive channels.

Transaction Level Modeling (TLM) is mainly used for early platform evaluation, performance analysis, and fast simulation of HW/SW systems. The general idea is to use *transactions* as an abstraction for all kind of data that is transmitted between different modules. This enables simulations on different abstraction levels, trading off accuracy and simulation speed. The TLM standard [20] and its implementation are an extension of SystemC, which provide interoperability between different transaction level models. The core of the TLM standard is the *interoperability layer*, which comprises standardized transport interfaces, sockets, and a generic payload.

2.2 Uppaal Timed Automata

Timed automata [1] are finite-state machines extended by clocks. Two types of clock constraints are used to model time-dependent behavior: *Invariants* are assigned to locations and restrict the time the automaton can stay in this location. *Guards* are assigned to edges and enable progress only if they evaluate to true. Networks of timed automata are used to model concurrent processes, which are executed with an interleaving semantics and synchronize on channels.

Uppaal [2] is a tool suite for modeling, simulation, and verification of networks of timed automata. The Uppaal modeling language extends timed automata by bounded integer variables, a template mechanism, binary and broadcast channels, and urgent and committed locations. Binary channels enable a blocking synchronization between two processes, whereas broadcast channels enable non-blocking synchronization between one sender and arbitrarily many receivers. Urgent and committed locations are used to model locations where no time may pass. Furthermore, leaving a committed location has priority over non-committed locations.

A small example Uppaal timed automaton (UTA) is shown in Figure 1. The initial location is denoted by ◎, and `request?` and `ack!` denote sending and receiving on channels, respectively. The clock variable `x` is first set to zero and then used in two clock constraints: the invariant `x <= maxtime` denotes that the corresponding location must be left before `x` becomes greater than maxtime, and the guard `x >= mintime` enables the corresponding edge at `mintime`. The symbols ⓤ and © depict urgent and committed locations.

2.3 Transformation from SystemC to UPPAAL

In previous work [13, 14, 21], we have presented an approach for the automatic transformation of the informally defined semantics of SystemC/TLM designs

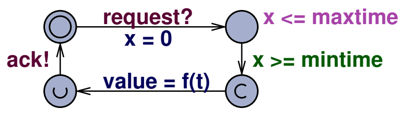

Fig. 1: **Example Timed Automaton**

into the formal semantics of UTA. The transformation preserves the (informally defined) behavioral semantics and the structure of a given SystemC design and can be applied fully automatically. It requires two major restrictions. First, we do not handle dynamic process or object creation. This hardly narrows the applicability of the approach, as dynamic object and process creation are rarely used in SystemC designs. Second, the approach only supports data types that can be mapped to (structs and arrays of) int and bool.

In our transformation, we use predefined templates for SystemC constructs such as events, processes and the scheduler. Then, each method is mapped to a single UTA template. Call-return semantics is modeled with binary channels. Process automata are used to encapsulate the method automata and care for the interactions with event objects, the scheduler, and primitive channels. Our transformation is compositional in the sense that we transform each module separately and compose the system in a final instantiation and binding phase. For detailed information on the transformation of SystemC/TLM designs to UTA we refer to [12].

3 Related Work

In the past ten years, there has been a lot of work on the development of formal memory models for C and C-like languages and in particular on the verification of pointer programs. Three main approaches to reason about memory in C (cf. [26]) exist: First, semantic approaches regard memory as a function from some kind of address to some kind of value. Second, there exist approaches that use multiple typed heaps in order to avoid the necessity of coping with inter-type aliasing. In these approaches, a separate heap is used for each language type that is present in a given program or design. In [5], Bornat describes under which restrictions such a memory model is semantically sound. Third, approaches based on separation logic (an extension of Hoare logic) [22] are able to cope with aliasing and frame problems. The main idea of separation logic is to provide inference rules that allow for the expression of aliasing conditions and local reasoning.

With our approach, we mainly adapt the idea of multiple typed heaps [5] by providing a separate memory array for each datatype used in a given design.

There also have been several approaches to provide a formal semantics for SystemC in order to enable automatic and complete verification techniques. However, many of them only cope with a synchronous subset of SystemC [18, 23, 24, 10], cannot handle dynamic sensitivity or timing, and do not consider pointers or memory. Other approaches which are based on a transformation from SystemC into some kind of state machine formalism [11, 25, 27, 19], process algebras [17, 9] or sequential C programs [6, 7] do not cope with pointers or memory as well. Furthermore, most of these approaches lack some important features (e.g., no support for time, no exact timing behavior, no automatic transformation). To the best of our knowledge, the only approach that can cope with pointers and memory is the work of [16, 4]. There, a labeled Kripke structure-based semantics for SystemC is proposed and predicate abstraction techniques are used for verification. However, the main idea of this approach is to abstract from the hardware by grouping it into combinational and clocked threads, which are then combined into a synchronous product for the overall system. They do neither address timing issues nor inter-process communication via sockets and channels. Thus, it remains unclear how they would cope with deeply integrated hardware and software components and their interplay. Several approaches exists for the verification of memory safety properties. However, these approaches focus on pure C (e.g., BLAST [3] and VCC/Z3 [8]) and cannot cope with the special semantics of SystemC/TLM.

4 Formalization and Transformation of the SystemC Memory Model

In this paper, we present a novel approach for model checking memory-related properties of HW/SW systems implemented in SystemC/TLM. The main idea of our approach is to formalize a clean subset of the SystemC memory model using separate memory arrays for each type present in a given design (cf. [5]). In order to enable model checking of memory-related properties, we incorporate this formal memory model into our SystemC to Timed Automata Transformation Engine (STATE) [13, 14, 21]. To this end, we define a set of transformation rules, which covers all memory-related constructs that are relevant for our subset of the SystemC memory model. For each memory-related construct, we define a UTA representation. With that, we can automatically transform a given SystemC/TLM design that makes use of pointers and memory into a UTA model.

In the following, we first state a set of assumptions that define a subset of the SystemC memory model. Then, we present our representation of the SystemC memory model within Uppaal. Finally, we present the transformation itself.

4.1 Assumptions

Our memory model covers many memory related constructs like call-by-reference of methods, referencing of variables, derefencing of pointers and pointers to pointers in arbitrary depth. However, we require a given SystemC/TLM model to fulfill the following assumptions:

1. No typecasts are used.
2. No direct hardware access of memory addresses (e. g., *int *p; p = 0xFFFFFF;*).
3. Structs are only referenced by pointers of the same type as the struct. This also means that there are no direct references to struct members.

4. No pointer arithmetic is used.
5. No dynamic memory allocation.
6. No recursion is used.
7. No function pointers are used.

Assumptions 1, 5, and 6 are necessary as Uppaal does not support typecasting, dynamic memory allocation or recursion. The second assumption is necessary because we do not model the memory bytewise and can only access it per variable. The third assumption is due to the fact that we do not flatten structs and therefore struct members do not have an own address. As we only model the data memory, assumption 7 is necessary. The Assumptions 1-4 can be considered as minor ones and hardly restrict the expressiveness of our memory model. As most SystemC/TLM models do neither make use of dynamic memory allocation nor of recursion, Assumptions 5 and 6 are acceptable as well.

With the assumptions above, we have a clear definition of the subset of the SystemC memory model that we want to support with our approach.

4.2 Representation

The main idea of our representation is to model the memory of the SystemC/TLM design with multiple typed arrays. As Uppaal does not support polymorphic datatypes, we create a separate array for each datatype used in the design. Pointers then can be modeled as integer variables, which point to a position in the array for the corresponding type. Array variables are interpreted as pointers to the first array element. All other variables are modeled as constant pointers if they are ever referenced (e.g., call-by-reference or direct referencing).

Figure 2 shows a small example of our Uppaal representation for the SystemC memory model. While pointers, integers and struct variables are arbitrarily spread over the memory in the SystemC memory model, we group them together in our Uppaal representation. In our example, there exist an integer variable `i` and two objects `s` and `t` of type `data`. Furthermore, there is an integer pointer `p`, pointing to `i`, and a pointer `q` of type `data`, pointing to `t`. In the resulting Uppaal model, `i` is placed in the array `intMem` and `s` and `t` are placed in the array `dataMem`. The pointers are transformed from real addresses into the corresponding array indices. Note that the arrays have a finite and fixed size which cannot be altered during the execution of the model. However, the pointers can point to all existing data of their type.

4.3 Transformation

For the transformation of a given design, we sort all variables into three disjunct sets: *PTR*, containing all pointers, *REF* containing all referenced nonpointer variables and all arrays and *VAR* containing all other nonpointer variables.

The result can be used to extract the memory model of the SystemC/TLM model and to transform it into a Uppaal memory model as proposed in 4.2. Figure 3 shows a small example for the transformation. Except for additional

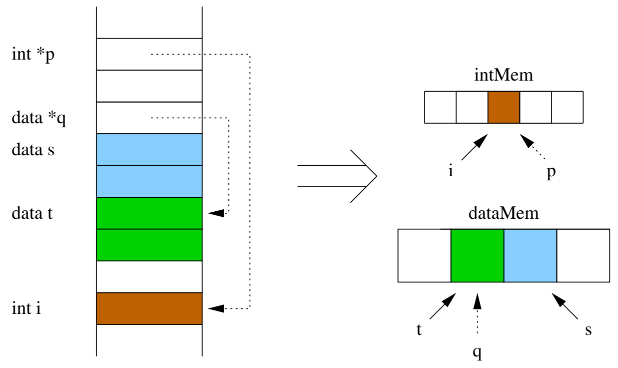

Fig. 2: Memory Representation in SystemC and UPPAAL

array accesses, the resulting UPPAAL model has the same structure and variable names as the original design. This eases manual matching with the corresponding SystemC/TLM design to correct detected errors in SystemC/TLM designs.

```

data arr[3];
data x;
data *p;

x.val = 23;
p = &x;
p->val = 42;
arr[2].val = 12;
*p = arr[1];
p = &arr[2];
```

(a) SystemC/TLM representation

```
int dataMEM[DATAMEMSIZE];
const int arr = 0;
const int x = 3;
int p;

dataMEM[x].val = 23;
p = x;
dataMEM[p].val = 42;
dataMEM[arr+2].val = 12;
dataMEM[p] = dataMEM[arr+1];
p = arr + 2;
```

(b) UPPAAL representation

Fig. 3: Memory Representation Examples

Table 1 shows the transformation rules we use, with var $\in REF$, arr $\in REF \wedge isArray(arr)$, p,q $\in PTR$, arbitrary data types T, U, V and W, and the arbitrary expression E. In general, every referenced variable is converted into a typed array index. While for nonpointer variables this index is constant, pointers can be arbitrarily changed. Direct accesses to these variables and pointer dereferencing operations can be modeled by typed array accesses. Direct pointer manipulation and variable referencing can be performed without any typed array access.

For all variable types in the REF and PTR sets, we generate a typed array representing the memory for this type. The size of each typed array is determined by the total amount of referenced variables of this type. For all members of the *REF* set, we reserve one field in the typed array per variable per module instance and generate a constant integer with the name of the variable and the index of the reserved field. For arrays, we reserve one field per element and set the constant integer to the index of the first element in the array. We replace every variable access with an array access to the typed array and every referencing of the variable by a direct access. If the variable is an array, the index is used as an offset. For all members of the *PTR* set we generate an integer variable with the initial value NULL (-1) or the index of the variable the pointer points to. As -1 is not a valid index of the typed array, all accesses to uninitialized pointers result in an array index error. Furthermore, we replace every dereferencing operation to the pointer with an array access to the corresponding typed array.

We implemented the transformation rules in our previously proposed transformation from SystemC to UPPAAL and thus can transform a given SystemC/TLM model with pointers fully automatically. Currently, our implementation does not support pointers to pointers and arrays of pointers, though both can be added with little effort.

Table 1: Transformation Rules

SystemC	UPPAAL
Declarations	
T var;	⇒ const int var = newIndex(T);
T var = E;	⇒ const int var = newIndex(T); TMEM[var] = E;
U arr[E];	⇒ const int arr = newIndex(U, E);
U arr[] = {v_0,...,v_{n-1}};	⇒ const int arr = newIndex(U, n); UMEM[arr+0] = v_0; ...; UMEM[arr+(n-1)] = v_{n-1};
V *p;	⇒ int p = -1;
W *q = E;	⇒ int q = E;
Variable Access	
var	⇒ TMEM[var]
arr[E]	⇒ UMEM[arr+E]
&(E)	⇒ E
&arr[E]	⇒ arr+E
Pointer Access	
*(E)	⇒ TMEM[E] (with E of type T)
NULL	⇒ -1
Field Access	
var.field	⇒ TMEM[var].field
var.p→field	⇒ VMEM[TMEM[var].p].field
arr[E].field	⇒ UMEM[arr+E].field
arr[E].p→field	⇒ VMEM[UMEM[arr+E].p].field
p→field	⇒ VMEM[p].field
p→q→field	⇒ WMEM[VMEM[p].q].field

5 Verification of Memory Safety

As our transformation from SystemC to Uppaal is able to cope with pointers and other memory-related constructs, the Uppaal model checker can now be used to verify memory safety properties. In general, we can verify all properties that can be expressed within the subset of CTL supported by Uppaal [2] (e.g., safety, liveness and timing properties as shown in [21]). For convenience, our verification framework generates two memory safety properties automatically:

(a) All pointers in the design are always either null, or they point to a valid part of the memory array corresponding to their type.
(b) The design never tries to access memory via a null pointer.

To verify the first property, it is necessary to check for all pointers $p_0 ... p_n$ that they are either null or have a value within the range of their typed array. If the function $u(p_i)$ yields the size of the typed array of the type of p_i, property (a) can be formalized as follows:

$$AG\ (p_0 = \text{null} \vee 0 \leq p_0 \leq u(p_0) - 1) \wedge\ ...\ \wedge\ (p_n = \text{null} \vee 0 \leq p_n \leq u(p_n) - 1)$$

The second property cannot be captured statically, as it needs the dynamic information where in the program a pointer is used to access memory. To solve this problem, we have developed an algorithm identifying all memory accesses in all processes $\text{Proc}_0 ... \text{Proc}_n$. For each transition comprising a memory access, a unique label l_i is assigned to its source location. With these labels, the property that a memory access ma_i that uses a pointer p_j is valid can be formalized as follows:

$$safe(ma_i) \equiv (\text{Proc}(ma_i).l_i \implies (p_j \neq \text{null}))$$

Using this abbreviation, the second property can be formalized as follows:

$$AG\ safe(ma_0) \wedge ... \wedge safe(ma_n)$$

Both memory safety properties described above are automatically generated for all pointers in a given design within our verification framework.

6 Evaluation

In this section we evaluate our approach with an industrial case study, namely a TLM implementation of the AMBA AHB, provided by Carbon Design Systems.

The original model consists of about 1500 LOC. To meet the assumptions of our approach, we performed the following modifications: (1) we changed the sockets to TLM standard sockets, (2) we replaced the generic payload type with a specific one, (3) we replaced operators for dynamic memory management (e.g., *new, delete*) by static memory allocation and (4) we only transfer constant data through the bus. The latter modification drastically simplifies the verification problem. However, our focus is on verifying the correct concurrent behavior,

Table 2: Results from the Amba AHB Design

	Verification time ([h:]min:sec)							
	Pointer-free design				Design with pointers			
	1M1S	1M2S	2M1S	2M2S	1M1S	1M2S	2M1S	2M2S
transformation time	0:04	0:04	0:04	0:04	0:03	0:03	0:04	0:05
deadlock freedom	< 1	< 1	0:27	1:08	6:17	12:06	37:28	1:24:25
only one master	-	-	0:14	0:34	-	-	24:07	54:32
bus granted to M1	< 1	< 1	0:15	0:37	3:56	7:47	24:10	54:06
bus granted to M2	-	-	0:15	0:38	-	-	24:10	54:02
timing	< 1	< 1	0:22	0:54	11:37	22:24	1:09:15	2:32:04
memory safety (a)	-	-	-	-	4:36	9:16	27:57	1:04:07
memory safety (b)	-	-	-	-	6:36	13:19	39:12	1:27:29
# states	5K	9K	537K	1M	15M	26M	64M	127M
memory usage	< 1 mb	2 mb	61 mb	112 mb	873 mb	1.5 gb	2.2 gb	3.9 gb

synchronization, timing, and memory safety which do not depend on the data that is transfered over the bus. The modified model consists of about 1600 LOC.

We also performed experiments on a pointer-free variant of the AMBA AHB design to evaluate the additional verification effort produced by our memory model. Therefore, we manually removed all memory related constructs from the original design and tried to keep the resulting design completely side-effect free. In the following, we compare the results of two different experiments: transformation and verification of (1) the pointer-free design and (2) of a design featuring pointers and other memory-related constructs (like call-by-reference).

For both designs, we verified the following properties: (1) deadlock freedom, (2) a bus request is always eventually answered with a grant signal, (3) the bus is only granted to one master at a time, (4) a transaction through the bus is always finished within a given time limit. For the CTL formulae we refer to [21]. For the design with pointers and other memory-related constructs, we additionally verified memory safety, as described in Section 5. All experiments were run on a 64bit Linux system with a dual core 3.0 GHz CPU and 8 GB RAM. To evaluate the scalability of our approach we used different design sizes (from 1 master and 1 slave, 1M1S, to 2 master and 2 slaves, 2M2S). The results of the verification are shown in Table 2.

All properties have been proven to be satisfied at the end of the verification phase. During the verification, we detected a bug in the original design which led to a deadlock situation. When a transaction is split into several separate transfers, a counter variable is used to store the number of successful transfers before the split occurs. This variable was not reset in the original design. As a consequence, all split transactions besides the first one failed. This is a typical example which is both difficult to detect and to correct with simulation alone. With our approach, the generation of a counter example took only a few minutes. Due to the structure preservation of our transformation and the graphical visualization in Uppaal, it was easy to understand the cause of the problem.

Our results show that the verification effort, in terms of CPU time and memory consumption, is drastically increased if pointers and other memory-related constructs are taken into account. This is due to the fact that the memory model introduces an additional integer variable for each variable in the design. However, formal verification via model checking, if successful, is only performed once during the whole development cycle. At the same time, the generation of counter examples only takes a few minutes. Most importantly, we are not aware of any other approach that can cope with the HW/SW interplay within SystemC/TLM models and at the same time facilitates the verification of memory-related properties, for example memory safety.

7 Conclusion and Future Work

We presented a novel approach for model checking of memory-related properties on HW/SW systems implemented in SystemC/TLM. We formalized a clean subset of the SystemC memory model with UTA. We use this formalization for a fully-automatic transformation of SystemC/TLM into equivalent Uppaal timed automata. This enables the use of the Uppaal model checker to verify memory-related properties. For convenience, we generate two memory safety properties, namely that all pointers only point to valid memory locations or null and that no null pointer accesses are used, automatically within our verification framework.

We implemented our approach and showed its applicability with an industrial design of the AMBA Advanced High Performance Bus (AHB). We were able to verify deadlock freedom, timing, and memory safety. We detected a deadlock situation in the AMBA AHB design, which could easily be resolved with the help of the counter-example generated by the Uppaal model checker. Our memory model produces a significant overhead to verification time and memory consumption. However, this overhead is compensated with the possibility to verify memory-related properties and the drastically increased practical applicability of our approach.

In our case study, we manually modified the design such that only constant data is transfered over the bus. For future work, we plan to extend our approach with automatic data abstraction techniques to enable the verification of even larger SystemC/TLM designs without manual interaction.

References

1. Alur, R., Dill, D.L.: A Theory of Timed Automata. Theoretical Computer Science 126, 183–235 (1994)
2. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on Uppaal. In: Formal Methods for the Design of Real-Time Systems. pp. 200–236. LNCS 3185, Springer (2004)
3. Beyer, D., Henzinger, T., Jhala, R., Majumdar, R.: Checking Memory Safety with Blast. In: Fundamental Approaches to Software Engineering, pp. 2–18. LNCS 3442, Springer (2005)
4. Blanc, N., Kroening, D., Sharygina, N.: Scoot: A Tool for the Analysis of SystemC Models. In: TACAS. pp. 467–470. LNCS 4963, Springer (2008)

5. Bornat, R.: Proving pointer programs in Hoare Logic. In: MPC. pp. 102 – 126. LNCS 1837, Springer (2000)
6. Cimatti, A., Micheli, A., Narasamdya, I., Roveri, M.: Verifying SystemC: A software model checking approach. In: FMCAD. pp. 51 –59 (2010)
7. Cimatti, A., Griggio, A., Micheli, A., Narasamdya, I., Roveri, M.: Kratos - A Software Model Checker for SystemC. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV, pp. 310–316. LNCS 6806, Springer (2011)
8. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A Practical System for Verifying Concurrent C. In: TPHOL. pp. 23–42. LNCS 5674, Springer (2009)
9. Garavel, H., Helmstetter, C., Ponsini, O., Serwe, W.: Verification of an industrial SystemC/TLM model using LOTOS and CADP. In: MEMOCODE. pp. 46–55. IEEE (2009)
10. Große, D., Kühne, U., Drechsler, R.: HW/SW Co-Verification of Embedded Systems using Bounded Model Checking. In: Great Lakes Symposium on VLSI. pp. 43–48. ACM Press (2006)
11. Habibi, A., Moinudeen, H., Tahar, S.: Generating Finite State Machines from SystemC. In: DATE. pp. 76–81. IEEE (2006)
12. Herber, P.: A Framework for Automated HW/SW Co-Verification of SystemC Designs using Timed Automata. Logos (2010)
13. Herber, P., Fellmuth, J., Glesner, S.: Model Checking SystemC Designs Using Timed Automata. In: CODES+ISSS. pp. 131–136. ACM press (2008)
14. Herber, P., Pockrandt, M., Glesner, S.: Transforming SystemC Transaction Level Models into UPPAAL Timed Automata. In: MEMOCODE. pp. 161 – 170. IEEE Computer Society (2011)
15. IEEE Standards Association: IEEE Std. 1666–2005, Open SystemC Language Reference Manual (2005)
16. Kroening, D., Sharygina, N.: Formal Verification of SystemC by Automatic Hardware/Software Partitioning. In: MEMOCODE. pp. 101–110. IEEE (2005)
17. Man, K.L.: An Overview of SystemCFL. In: Research in Microelectronics and Electronics. vol. 1, pp. 145– 148 (2005)
18. Müller, W., Ruf, J., Rosenstiel, W.: SystemC: Methodologies and Applications, chap. An ASM based SystemC Simulation Semantics, pp. 97–126. Kluwer Academic Publishers (2003)
19. Niemann, B., Haubelt, C.: Formalizing TLM with Communicating State Machines. Forum on specification and Design Languages (2006)
20. Open SystemC Initiative (OSCI): TLM 2.0 Reference Manual (2009)
21. Pockrandt, M., Herber, P., Glesner, S.: Model Checking a SystemC/TLM Design of the AMBA AHB Protocol. In: ESTIMedia. pp. 66 – 75. IEEE (2011)
22. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. pp. 55–74. IEEE Computer Society (2002)
23. Ruf, J., Hoffmann, D.W., Gerlach, J., Kropf, T., Rosenstiel, W., Müller, W.: The Simulation Semantics of SystemC. In: DATE. pp. 64–70. IEEE (2001)
24. Salem, A.: Formal Semantics of Synchronous SystemC. In: Design, Automation and Test in Europe (DATE). pp. 10376–10381. IEEE Computer Society (2003)
25. Traulsen, C., Cornet, J., Moy, M., Maraninchi:, F.: A SystemC/TLM semantics in Promela and its possible applications. In: SPIN. pp. 204–222. LNCS 4595, Springer, Berlin (2007)
26. Tuch, H.: Formal Memory Models for Verifying C Systems Code (2008)
27. Zhang, Y., Vedrine, F., Monsuez, B.: SystemC Waiting-State Automata. In: Proceedings of VECoS 2007 (2007)