Functional Validation of System Level Static Scheduling

Samar Abdi
Center for Embedded Computer Systems
University of California, Irvine, USA
sabdi@cecs.uci.edu

Daniel Gajski
Center for Embedded Computer Systems
University of California, Irvine, USA
gajski@cecs.uci.edu

Abstract

Increase in system level modeling has given rise to a need for efficient functional validation of models above cycle accurate level. This paper presents a technique for comparing system level models, before and after the static scheduling of tasks on processing elements of the architecture. We derive a graph representation from models written in system level design languages (SLDLs) and define their execution semantics. Notion of functional equivalence of system level models is established using these graphs. We then present well defined rules for reduction of such graphs to a normal form. Finally, we show how to check for functional equivalence of two system level models by isomorphism of their normal graph representations. A checker built on the above concept is used to automatically validate the functional correctness of the static scheduling step. As a result, the models generated for various scheduling decisions do not have to be reverified using costly simulations.

1. Introduction

System level (SL) design has received much attention lately due to the rising complexity of modern HW-SW designs. Design methodologies now involve several modeling stages to take an executable specification of the design to a cycle accurate implementation in a gradual, step wise fashion. At each step, the system model is transformed to reflect the design decision made at that step. However, it is imperative that the functionality of the original specification is preserved as the design progresses through these incremental model transformations. In other words, we need to validate if two models, before and after the implementation of a design decision, are functionally equivalent. In this paper, we look at functional validation of model transformations resulting from system level static scheduling.

A possible system level design methodology is as follows. We start by distributing the specification tasks (referred to as **behaviors**) onto different HW and SW processing elements (PEs) to derive an architecture model. However, the behaviors in this architecture model are not yet scheduled. The static scheduling step allows for serializing the concurrent behaviors on the HW PEs, since they will be implemented with a single controller. Also, at this stage, the communication between PEs may be statically scheduled to optimize timing. Communication synthesis adds bus architecture and arbitration policy, resulting in a completely scheduled bus transaction model. Finally, the SW tasks are compiled for the target processor and the HW behaviors are synthesized.

Research on task level static scheduling has mostly been done for embedded systems targeting real time applications [9]. For distributed systems, static communication scheduling [8] has been proposed for improving timing. SLDLs like SystemC 2.0 [1] and SpecC [4] help in incorporating task level static scheduling in system level design methodologies by allowing the exploration of different schedules for both computation and communication in a much speedier manner than cycle accurate co-simulation. This has led to research being directed towards modeling and verification at system level in order to verify the correctness of design steps. Traditional software model checking [6] and bounded model checking [3] allow property verification of high level models written in C-like languages. However, to the best of our knowledge, there has been little work in equivalence verification of system level models. The closest work [7] relies on textual comparison of models and requires them to be syntactically very similar.

The rest of the paper is organized as follows. In Section 2, we look at how and why is static scheduling performed in SL models. Section 3 presents a graph based abstraction of SL models using key language concepts. We also look at the execution semantics of such graphs and their extraction from SLDL code. In Section 4, we propose a method for checking functional equivalence of SLDL models by reducing their abstracted graphs to a normal form. The rules for performing this graph normalization are also explained. Finally, we present experimental results for a functional equivalence checker based on the above concept and wind up with conclusions.

2. Static scheduling in SL models

Static scheduling is performed in system level models either due to resource constraints or timing optimization. Behaviors mapped to HW are implemented using controller(s) and data path(s). As a result, behaviors that are grouped for implementation with the same controller must be serialized. Consider an unscheduled HW PE with two threads of execution. The first thread executes behavior b_1 followed by b_2, while the second thread executes b_3 followed by b_4. A possible serialization of the PE would sequentially execute the behaviors in the order $\{b_1, b_3, b_2, b_4\}$. Other schedules, that do not violate data dependencies, are also possible.

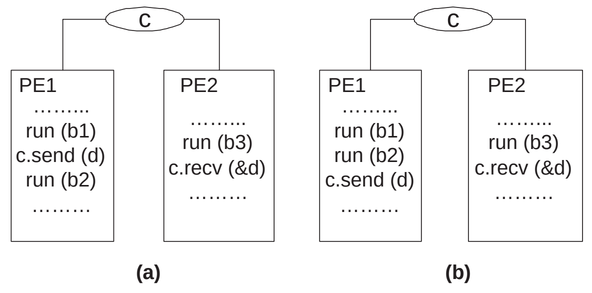


Figure 1. Different communication schedules for transaction over channel *c*.

Reordering of behaviors can also take place as a result of communication scheduling. Such a scenario is shown in Figure 1, where data *d* is sent from PE1 to PE2 over channel *c*. The channel implements rendezvous communication semantics, i.e. both sender and receiver must synchronize for the transaction to take place. Consequently, for the case shown in Figure 1(a), b_2 must wait until b_3 has completed and the transaction is performed. If b_3 takes a long time to execute, execution inside PE1 will stall, as it waits for the data transaction. Behavior b_2 may be scheduled before the transaction, if it has no data dependency on b_3. The resulting schedule,shown in 1(b), optimizes timing.

3. Graph abstractions of SL models

Computation in SLDLs is encapsulated inside behaviors that read data from variables via in-ports, perform local computation, and then write data to variables via outports. Most SLDLs, also support the concept of hierarchy, where a complex behavior can be described in terms of sub-behaviors and their compositions. A behavior without any sub-behaviors is called a *leaf behavior*. We will assume that model transformations will treat the leaf behaviors as atomic. We also define a class of leaf behaviors, B^I, consisting of *identity* behaviors that output the same data as their input. They do not perform any computation, and are typically used as place holders or for data routing. We assume channel transactions only between identity behaviors.

In this section we will define two graphs, namely the *behavior control graph* (BCG) and the *port connection network* (PCN). The former is used to capture the control dependencies between leaf behaviors, while the latter captures data dependencies between various objects in the model.

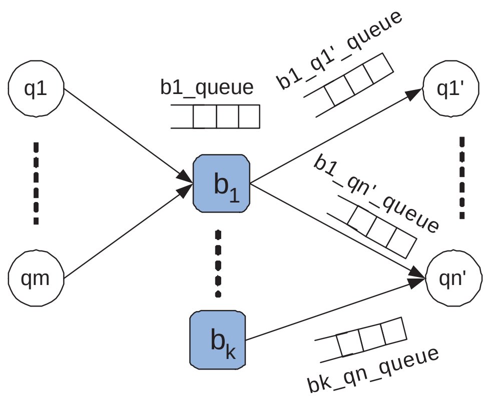


Figure 2. The firing semantics of BCG nodes

3.1. Control and data dependency graphs

The BCG is a directed graph (N,E) with two types of nodes, namely *behavior nodes*(N_B) and *control nodes*(N_Q). The behavior nodes, as the name suggests, indicate behavior execution, while the control nodes evaluate control conditions that lead to further behavior executions. Directed edges are allowed from behavior nodes to control nodes and vice versa. Also, a control node can have one, and only one, out going edge. Thus,
$E(BCG) \subset N_B(BCG) \times N_Q(BCG) \cup N_Q(BCG) \times N_B(BCG)$

The execution of a behavior node, and similarly, evaluation in a control node, will be referred to as a *firing*. Node firings are facilitated by tokens that circulate in the queues of the BCG as shown in Figure 2. Each behavior node (shown by rounded edged box) in the BCG has one queue, for instance *b*1_*queue* for behavior node b_1. All incoming edges to a behavior node represent the various writers to the queue. A behavior node blocks on an empty queue and fires if there is at least one token in its queue. Upon firing, one token is dequeued from the node's queue. The control node (shown by circular node), on the other hand, has as many queues as the number of incoming edges. For instance q_n has k queues, one each for edges from b_1 through b_k. A control node checks all its queues and blocks on empty queues.

If the queue is not empty, it dequeues a token from the queue and proceeds to check the next queue. The node fires after it has dequeued one token from each of its queues.

After firing, a behavior node generates as many tokens as its out-degree, and each token is written to the corresponding queue of the destination control node in a non-blocking fashion. Upon firing, the control node evaluates its condition. If the condition evaluates to TRUE, then a token is enqueued to the queue of the destination behavior node.

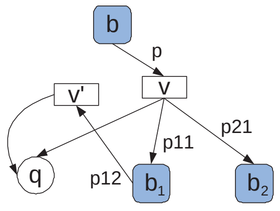


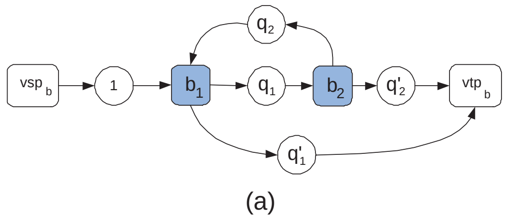
Figure 3. Port connection network showing data dependencies

The PCN is a directed graph which has three types of nodes, namely behavior nodes (N_B), condition nodes (N_Q) and variable nodes (N_V). The edges represent data dependencies in the model and are labeled using the port names involved in the dependency as shown in Figure 3. For instance, a directed edge from a behavior node b to a variable node v (shown by rectangular box), labeled p (written (b, v, p)) would mean that b writes to the storage indicated by v via its out-port p. Similarly, an edge from a variable node v' to a behavior node b', labeled p' (written (v', b', p')) would indicate that b' reads variable v' using its in port p'. Note that for each variable v, there can be only one writer behavior (written as $wr(v)$). Control conditions also create data dependencies in the model. Thus, if a control condition q is a boolean function call $q = f_b(v_1, v_2, ..., v_n)$, then the node representing q has a directed edge from all the n variable nodes v_1 thorough v_n.

3.2. Deriving BCG and PCN from SLDL models

BCG and PCN are powerful enough to represent a model's execution trace at the granularity level of leaf behaviors. However, they lack the concept of hierarchy that most SLDLs have. Also, BCG does not have different constructs for sequential and concurrent execution. Concurrency is realized simply by orthogonality of the behavior nodes. We will now show how a model written in a SLDL can be abstracted into a BCG, PCN pair.

The control flow between behaviors is typically expressed using switch-case or goto constructs in SLDL. A simple pseudo code example for a hierarchical behavior b is as follows



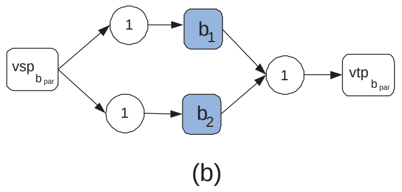


Figure 4. BCGs for different hierarchical behavior compositions

```
l1: **run** $b_1$; **if** $q_1 == 1$ **goto** l2 **else** break;
l2: **run** $b_2$; **if** $q_2 == 1$ **goto** l1 **else** break;
```

This behavioral composition is illustrated in Figure 4(a). Note the addition of placeholder identity behaviors vsp_b and vtp_b (represented by hollow boxes). The former indicates the *virtual starting point* of b, while the latter indicates the *virtual terminating point* of b. The addition of these identity behaviors makes the BCG polar, which helps in flattening the model. Therefore, any control dependency leading to b can be realized in BCG by an edge leading to vsp_b. Similarly, any control dependency from b can be represented by an edge from vtp_b in the BCG.

The resolution of parallel compositions is done similarly, as shown in Figure 4(b). The SLDL statement for a parallel composition: **par** $\{$**run** b_1; **run** $b_2\}$ creates a hierarchical behavior b_{par}. Execution of b_{par} indicates that both b_1 and b_2 are ready to execute. The execution of b_{par} terminates when both b_1 and b_2 have terminated. Again, $vsp_{b_{par}}$ and $vtp_{b_{par}}$ serve as the starting and terminating points, respectively, of the hierarchical behavior.

Due to their rendezvous semantics, channels in SLDLs imply control flow dependencies between communicating behaviors. Figure 5 demonstrates this control dependency extraction from channels. Recall our assumption that the SLDL model has channel transactions only between identity behaviors. The synchronization properties of the SLDL channel would ensure that any behavior following the sender identity behavior would not execute until the receiver identity behavior has executed, and vice versa. If we were to optimize away the channel to reveal the control dependencies, the result will be as shown in figure 5. Note that the channel also results in the edge (e, e') in the PCN.

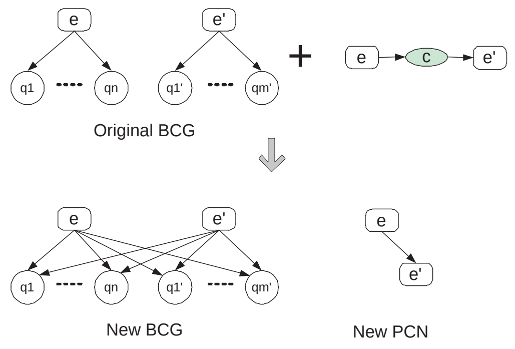


Figure 5. Effect of rendezvous channel on BCG and PCN

4. Equivalence checking of SL models

In this section we present methods for automatically checking if two models, each represented by a BCG, PCN pair, are functionally equivalent. We define a notion of equivalence and rules for reducing the graphs to an equivalent normal form. If the normal forms for two models are identical, then they are functionally equivalent.

4.1. Notion of Functional Equivalence

Our notion of functional equivalence is based on the trace of values that the variables hold during model execution. In particular, we are interested in the variables that are written to by non-identity behaviors. We will refer to such variables as *observed* variables. The reasoning is that variables that are connected to the output ports of identity behaviors are simply a copy of another variable. Informally speaking, we consider two models to be functionally equivalent, if they have identical observed variables and the trace of values assumed by those variables during model execution is identical, given the same initial assignment. The formal notion of equivalence is as follows.

Given a model M, let $I(M)$ be the initial assignment of observed variables in M. Let
$\forall v \in N_V(PCN(M)), \exists wr(v) \in N_B(PCN(M))$
Let $d_i, i > 0$ be the value written to v after the i^{th} execution of $wr(v)$. Let d_0 be the initial assignment value of v. We define the ordered set
$\tau(v, M, I(M)) = \{d_0, d_1, d_2, ...\}$
We claim that two models M and M' are equivalent iff
$\forall v, I(M) = I(M') \Rightarrow \tau(v, M, I(M)) = \tau(v, M', I(M'))$

4.2. Graph reduction

Our goal is to eliminate identity behavior nodes and redundant dependencies from the BCG and PCN, as the model is reduced to its normal form. Redundant dependencies include control dependencies that do not influence the value trace of the observed variables or set of dependencies that may be replaced by a smaller set.

4.2.1 Identity Elimination

The identity behavior, by definition, does not perform any computation. Hence, we may remove the identity behaviors from BCG and PCN, while making appropriate changes to the variable dependencies.

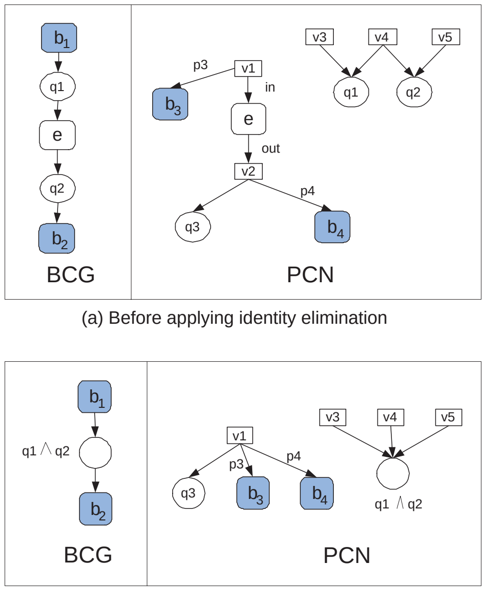

b1
q1
e
q2
b2
BCG
PCN
p3
v1
in
b3
out
v2
p4
q3
b4
v3
v4
v5
q1
q2
(a) Before applying identity elimination
q1 ∧ q2
(b) After applying identity elimination

Figure 6. Parts of BCG and PCN before and after identity elimination

The simple example illustrated in figure 6(b) shows parts of the BCG and PCN involving an identity behavior e. In the BCG, e is part of the control path from b_1 to b_2. It must be noted that there are no other edges to either e or the control nodes q_1 and q_2. As per the semantics of BCG, we can eliminate e by merging the control nodes q_1 and q_2 as shown for the BCG in figure 6(b). Note that in both the models, b_2 will execute after b_1 if both control conditions q_1 and q_2 evaluate to TRUE. Hence, the elimination of e leads to the merging of nodes q_1 and q_2 to form the new control node labeled as $q_1 \wedge q_2$ (ANDing of the boolean variables q_1 and q_2). However, it must be noted that as a result of elimination of e, the variable that e was writing to, also becomes invalid. This variable v_2 is shown in the PCN in figure 6(a). Now, variable v_2 is simply a copy of v_1, by definition of the identity behavior. Therefore, all dependencies on v_2, including in-port connections for behaviors and parameters for control conditions, must be replaced by dependencies on v_1. The elimination of e from the original model results in the PCN shown in figure 6(b). This simple example of identity elimination shows how the reduction rule works in principle. We now present the general definition of the rule.

Identity Elimination Rule: Given a model M, let $e \in N_B(M)$ be an identity behavior. Let M' be the model resulting from elimination of e. Let there be m edges to e from control nodes q_1 through q_m in BCG(M). Also, let there be n edges from e to control nodes labeled q'_1 through q'_n in BCG. Now, $\forall i, j, s.t. 1 \leq i \leq m, 1 \leq j \leq n$
In BCG(M), q_i has in-degree $l(i)$ and q'_j has in-degree $k(j)+1$.
Let, $(x^i_1, q_i), (x^i_2, q_i), ..., (x^i_{l(i)}, q_i) \in E(BCG(M))$, and $(e, q'_j), (y^j_1, q'_j), ..., (y^j_{k(j)}, q'_j) \in E(BCG)$ Also, let $(q'_j, z_j) \in E(BCG)$. After, elimination of e, the merger of control nodes would result in $m \times n$ new control nodes. Therefore, $\forall i, j, s.t. 1 \leq i \leq m, 1 \leq j \leq n$
$q_i \wedge q'_j : x^i_1 \& ... \& x^i_{l(i)} \& y^j_1 \& ... \& y^j_{k(j)} \rightsquigarrow z_j \in BCG(M')$
In the PCN, if $(e', e), (e, v, out) \in PCN(M), e' \in B^I$, then
$PCN(M') = (PCN(M) - (e', e)) \cup (e', v, out)$.
If $(v, e, in), (e, v', out) \in PCN(M)$,
then $\forall x$, s.t.$(v', x, p) \in PCN(M)$
$PCN(M') = (PCN(M) - (v', x, p)) \cup (v, x, p)$.

4.2.2 Redundant control dependency elimination

In order to eliminate spurious edges in a BCG, we first need a control dependence analysis. Given model M, let $y \in N_B(BCG(M)), x \in N(BCG(M))$. If during any execution of M, y **always** fires at least once before and at least once between every firing of x, then we define y to be a **dominator** of x. The set of dominator nodes for x will be represented by $dom(x, M)$. The set $dom(x, M)$ can be defined inductively as follows

1. If $x \in N_B(BCG)$, then $dom(x, M) = dom(x, M) \cup \bigcap_{(q,x) \in E(BCG(M))} \{y : y \in dom(q, M)\}$

2. If $x \in N_Q(BCG)$, then $dom(x, M) = dom(x, M) \cup \bigcup_{(b,x) \in E(BCG(M))} \{b \cup \{y : y \in dom(b, M)\}\}$

Given $q \in N_Q(BCG(M))$. Let
$b_1, b_2 \in N_B(BCG)$ and $(b_1, q), (b_2, q) \in E(BCG)$
Thus b_1 and b_2 must fire for q to fire.If we can show that $b_1 \in dom(b_2, M)$ then the edge (b_1, q) can be eliminated from the BCG. This is because, upon execution of b_1, a token will be enqueued in the queue corresponding to (b_1, q). Now, if b_2 executes, we know that b_1 has already executed and enqueued the relevant token. The node q will dequeue this token from b_1 and will wait for a token from b_2. Hence, a token from b_2 means that b_1 must already have a token sent to q. If we remove edge (b_1, q), while keeping edge (b_2, q), the order of firings in BCG would not change.

Control Dependency Elimination Rule: Given model M, let $q \in N_Q(BCG(M))$. If $\exists b_1, b_2 \in N_B(BCG(M))$, s.t. $b_1 \in dom(b_2, M)$ and $(b_1, q), (b_2, q) \in E(BCG(M))$, then $E(BCG(M)) = E(BCG(M)) - (b_1, q)$.

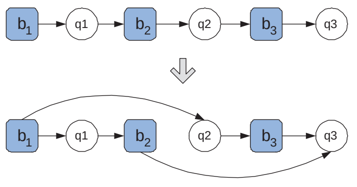


Figure 7. Control relaxation for edge (b_2, q_2)

4.2.3 Control Relaxation

Given model M, let $(b_2, q_2), (q_2, b_3) \in BCG(M)$. If there is no data dependency between b_2 and b_3 and between b_2 and q_2, then changing the order of firing between b_2 and q_2, or b_2 and b_3 would not change the value trace for any variable in M. Therefore, the artificial control dependency from b_2 to q_2 may be removed, as illustrated in figure 7. However, the rule applies only if the nodes q_1 and b_2 must have an in-degree of 1, while the node b_3 has an out-degree of 1. With these restrictions, $dom(b_2, M) = b_1 \cup dom(b_1, M)$. Thus, firing of b_1 will enqueue a token on the queue for b_3 if q_2 is TRUE. Also, the token released by firing of b_2 must be enqueued to q_3 if the edge (b_2, q_2) is to be removed. Hence, the transformation illustrated in figure 7 is functionally correct under the given restrictions.

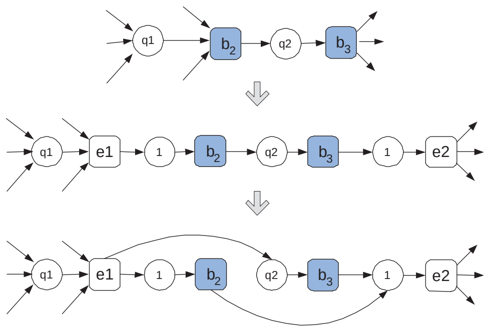


Figure 8. Control relaxation for edge (b_2, q_2) without in and out-degree restrictions

Control relaxation can be further generalized by removing the restrictions on the in-degree of b_2 and q_1 and the out-degree of b_3. The original BCG, with arbitrary degrees for the relevant nodes can be transformed as shown in figure 8. Using the inverse of rule on identity elimination, we can add identity behaviors e_1 and e_2 before b_2 and after b_3, respectively. This would allow us to use the control relaxation transformation to derive the BCG shown in the midle of figure 8. Finally, after control relaxation, the identity reduction rule can be applied to optimize away e_1 and e_2.

4.3. Testing functional equivalence

In order to validate functional equivalence of M and M', we convert their BCG and PCN to the normal form. The normal form of *M* is derived by iteratively applying the reduction rules to the BCG(M), PCN(M) pair until none of the rules is applicable anymore. The resulting normal form graphs are represented by NBCG(M) and NPCN(M). Similarly, we derive the normal form graphs for M'. If NBCG(M) is identical to NBCG(M') and NPCN(M) is identical to NPCN(M'), then M is equivalent to M'. This follows from transitivity of the equivalence relation and the functionality preserving nature of the reduction rules. Since, the static scheduling process (Section 2) changes only the control dependencies, the leaf level behaviors in both models should match. For lack of space, we could not present a walk through example of the normalization process. We refer the reader to [2] for the same.

Table 1. Performance of equivalence checker for different scheduling decisions

Sched. Type	Extraction Time	Reduction Time
serial	2.12s	5.66s
serial	2.11s	5.18s
comm.	2.42s	7.11s
comm.	2.83s	7.03s
comm.	3.06s	9.02s

5 Experimental Results

A tool, consisting of two modules, was written in C++ for checking equivalence of scheduled and unscheduled SpecC models. The graph extractor module derives the BCG and PCN from a SpecC Model, while the Graph Reducer module used the rules in Section 4.2 to generate the normalized form BCG and PCN. An automatic static scheduler was used to create models for different serializations on HW PEs. Experiments for communication scheduling were performed by manual transformation of models.

The model used is a GSM voice codec application [5] for cellular phones. The unscheduled model consisted of 1273 lines of SpecC code, with 43 non-identity leaf level behaviors that were distributed on 3 PEs. The BCG and PCN extraction and reduction times were in the order of a few seconds on a 2 GHz PC running RedHat Linux OS as shown in Table 1. The type column indicates the type of scheduling performed; either computation serialization (serial) or communication scheduling (comm.). Communication scheduling was done on the serialized model. The extraction time also includes the time for dominator analysis of all the nodes in the BCG. As expected, extraction and reduction of models after communication scheduling took slightly longer due to extra control dependencies and extra control relaxation cycles needed for normalization. Comparison of normalized graphs took negligible time.

6 Conclusion and Future Work

We presented a technique to check the functional equivalence of system level models before and after the scheduling of behaviors in the architecture PEs. The main advantage of this technique is that the scheduled model does not require any functional simulation. On the flip side, the equivalence checker cannot handle run-time scheduling that adds an RTOS model, which cannot be resolved by the checker. However, this is only to be expected since dynamic scheduling requires analysis at a smaller granularity level than the leaf level behaviors, for instance interleaving of concurrent behaviors. In the future, we would like to extend our equivalence checker to validate more design steps like communication synthesis.

References

[1] SystemC, OSCI[online]. Available: http://www.systemc.org/.

[2] S. Abdi and D. Gajski. System Level Verification with Model Algebra. Technical Report ICS-TR-04-29, University of California, Irvine, October 2004.

[3] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *TACAS 2004*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

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

[5] A. Gerstlauer, S. Zhao, and D. Gajski. Design of a GSM Vocoder using SpecC Methodology. Technical Report ICS-TR-99-11, University of California, Irvine, February 1999.

[6] G. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5), June 1997.

[7] H. Saito, T. Ogawa, T. Sakunkonchak, M. Fujita, and T. Nanya. An equivalence checking methodology for hardware oriented c-based specifications. In *IEEE International High Level Design Validation and Test Workshop*, pages 274–277, October 2002.

[8] D. Surma and E. Sha. Collision graph based communication scheduling for parallel systems. *International Journal of Computers and Their Applications.*, 5(1), March 1998.

[9] C. Wong, F. Thoen, F. Catthoor, and D. Verkest. Static task scheduling of embedded systems. In *3rd Workshop on System Design Automation - SDA 2000 Rathen, Germany*, pages 23–30, March 2000.