

# Model Validation for Mapping Specification Behaviors to Processing Elements

Samar Abdi and Daniel Gajski  
Center for Embedded Computer Systems  
University of California, Irvine, CA - 92697  
{sabdi,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 checking functional equivalence of system level models, before and after the distribution of behaviors in the specification over components in the platform architecture. We derive a control flow graph from models written in system level design languages (SLDLs) and reduce it to a normal form representation using well defined rules. Two models having identical normal form are shown to be functionally equivalent. An equivalence checker based on the above concept is used to automatically check if the architecture level model is functionally equivalent to the specification model. As a result, the models generated for various mapping decisions do not have to be reverified using costly simulations.

## 1 Introduction

System level design languages (SLDLs) are increasing being used to create high level system models. These models are typically constructed using a hierarchical composition of tasks, referred to as *behaviors*. System level design decisions are used to map a purely functional specification onto an architectural platform of processing elements (PEs), memories and communication elements. The models reflecting these design decisions need to be compared against the specification model to see if the two are functionally equivalent. In this paper, we will concern ourselves with the design decision of distributing behaviors in the specification model over different PEs in the architecture.

Figure 1 shows the methodology for generating the architecture model from the specification and checking the functional equivalence of the two models. The model generation algorithm uses the behavior mapping decision and syntactically transforms the specification model. The transformation creates a new hierarchy of behaviors, distinctly showing the top level PEs and the behaviors executing in them. Communication elements, called channels, are added

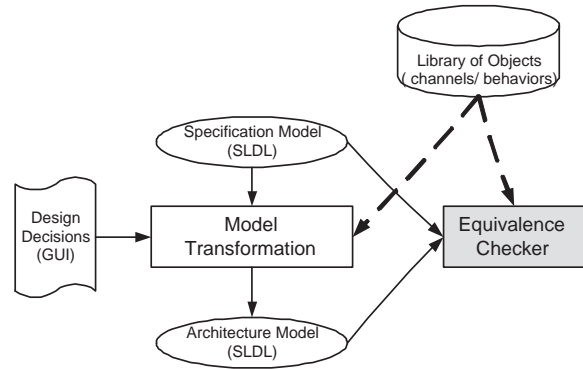


Figure 1. Automatic equivalence checking of system level models

to introduce synchronization between concurrently executing PEs and new behaviors are added to route data across PEs. These new channels and behaviors are obtained from a library of modeling objects. Since we need to compare the execution of the two models, the equivalence checker needs to resolve these library elements.

Some research has been done for comparison of SLDL models using textual correlation and symbolic simulation [7], but their approach requires two models to be very similar. Verification of only the synchronization primitives of SpecC [3] are presented in [8]. Correct by construction approaches have been proposed for HW/SW partitioning [2] and model generation [1], but they restrict the designer to follow a given refinement algorithm.

The rest of the paper is organized as follows. Section 2 shows how we use graphs to capture control flow in SLDL models. In this section, we also define our notion of equivalence and show how these graphs can be reduced to a normal form using well defined rules. Architecture model generation is illustrated with a simple example in Section 3. In Section 4, we use this simple example to illustrate how the reduction rules in Section 2 are used to check the functional equivalence of specification and architecture models. Ex-

perimental results with our equivalence checking algorithm are shown in Section 5. Finally, we wind up with conclusions and future work.

## 2 Graph Abstractions of SLDL 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 out-ports. We will assume that each variable is written to by only one behavior. We will refer to the writer of a given variable  $v$  as  $wr(v)$ . 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, called *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. We use a graph based data structure, namely the *behavior control graph* (BCG) to capture the control dependencies between leaf behaviors in the model.

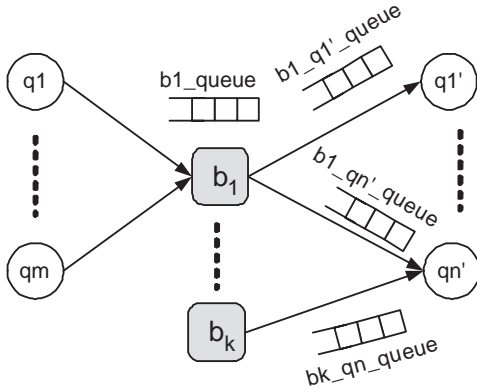


Figure 2. The firing semantics of BCG nodes

### 2.1 Behavior Control Graph

The BCG is similar in principle to the Kahn Process Network [5], but with some remarkable differences. It 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, outgoing 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  $b1\_queue$  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, sequentially 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 generated and written to the queue of the destination behavior node, else no token is generated.

### 2.2 Deriving BCG from SLDL Models

BCG is powerful enough to represent a model's execution trace at the granularity level of leaf behaviors. However, it lacks 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 control flow of a model written in a SLDL can be abstracted into a BCG.

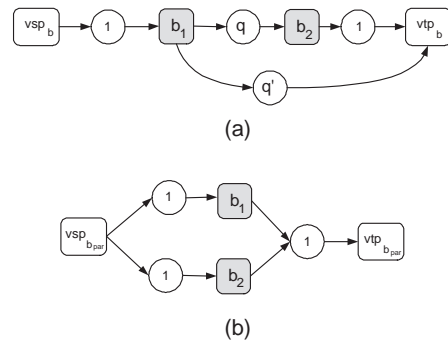


Figure 3. BCGs for different hierarchical behavior compositions

The control flow between behaviors is typically ex-

pressed using switch-case or goto constructs in SLDL. A simple pseudo code example for a hierarchical behavior  $b$  is as follows

$b$ : **run**  $b_1$ ; **if**  $q == 1$  **run**  $b_2$  **else break**;

This behavioral composition is illustrated in Figure 3(a). Note the addition of placeholder identity behaviors  $vsp_b$  and  $vt p_b$  (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  $vt p_b$  in the BCG.

The resolution of parallel compositions is done similarly, as shown in Figure 3(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  $vt p_{b_{par}}$  serve as the starting and terminating points, respectively, of the hierarchical behavior.

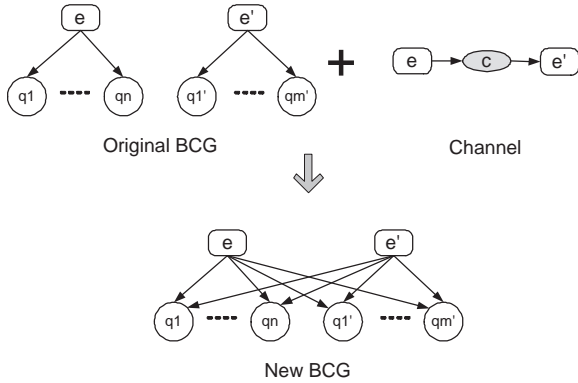


Figure 4. Effect of rendezvous channel on BCG

Due to their rendezvous semantics, channels in SLDLs imply control flow dependencies between communicating behaviors. Figure 4 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 4.

### 2.3 Notion of Functional Equivalence

Our notion of functional equivalence is based on the trace of values that the variables hold during model exe-

cution. 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

$$\forall v, \exists wr(v) \in N_B(BCG(M))$$

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

$$\tau(v, M, I(M)) = \{d_0, d_1, d_2, \dots\}$$

We claim that two models  $M$  and  $M'$  are equivalent iff

$$\forall v, I(M) = I(M') \Rightarrow \tau(v, M, I(M)) = \tau(v, M', I(M'))$$

From the above definition of functional equivalence, we have the following implications. Clearly, for two models to be equivalent, they should have a one-to-one mapping of non-identity leaf behaviors. Also, the control flow relations amongst these leaf behaviors must be identical. In particular, a functionality preserving transformation is one that does not alter the firing sequence of non-identity behavior nodes in the BCG under any initial assignment. We will now look at transformations to the BCG that retain the functionality of the model.

### 2.4 Graph Reduction

A given BCG by eliminating identity behavior nodes and redundant control dependencies. In this section, we will define the rules for reducing a BCG. It must be noted that the transformations applied in the reduction rules will preserve the original functionality of the model. If a BCG cannot be reduced any further by applying the reduction rules, then it is said to be in a *normal form*.

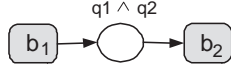
#### 2.4.1 Identity Elimination

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

The simple example illustrated in figure 5(a) shows part of a BCG involving an identity behavior  $e$  in 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 5(b). Note that in both the models,  $b_2$  will execute after  $b_1$  if both



(a) BCG before identity elimination



(b) BCG after identity elimination

Figure 5. BCG before and after identity elimination

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  $q_2$ ). 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 (R1):** Given a model  $M$ , let  $e \in N_B(BCG(M))$  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  $BCG(M)$ . 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  $k(j) + 1$ .

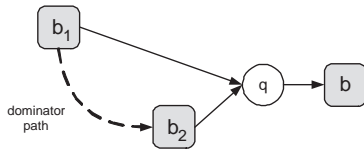
Let,  $(x_1^i, q_i), (x_2^i, q_i), \dots, (x_{l(i)}^i, q_i) \in E(BCG(M))$ , and

$(e, q'_j), (y_1^j, q'_j), \dots, (y_{k(j)}^j, q'_j) \in E(BCG)$  Also, let  $(q'_j, z_j) \in E(BCG)$ .

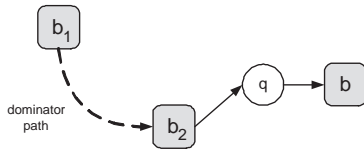
After, elimination of  $e$ , the merger of control nodes would result in  $m \times n$  new control nodes. Therefore,

$\forall i, j, s.t. 1 \leq i \leq m, 1 \leq j \leq n$

$q_i \wedge q'_j : x_1^i \& \dots \& x_{l(i)}^i \& y_1^j \& \dots \& y_{k(j)}^j \rightsquigarrow z_j \in BCG(M')$



(a) BCG before redundant control elimination



(b) BCG after redundant control elimination

Figure 6. BCG before and after redundant control elimination

## 2.4.2 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  $dom(x, M) = dom(x, M) \cup \bigcup_{(b,x) \in E(BCG(M))} \{b \cup \{y : y \in dom(b, M)\}\}$

An instance of control dependency elimination is shown in Figure 6. 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 (R2):** 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  $E(BCG(M)) = E(BCG(M)) - (b_1, q)$ .

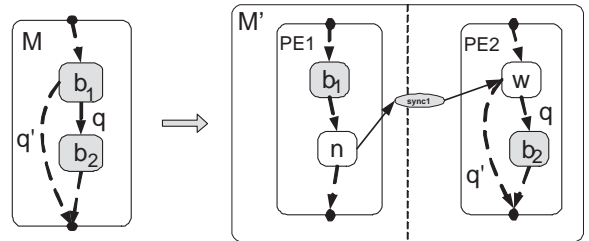


Figure 7. Model generation after behavior partitioning

## 3 Behavior Partitioning

Figure 7 shows a simple specification model  $M$  on the LHS with two behaviors  $b_1$  and  $b_2$  and condition control flow. After the execution of  $b_1$ , if condition  $q$  evaluates to TRUE, then  $b_2$  is executed, else the execution terminates.

The functionality of the specification model is same as that shown in the pseudo code in Section 2.2. Hence, the BCG for this model can be seen in Figure 3(a). On the RHS, we see an architecture level implementation  $M'$  where  $b_1$  is assigned to  $PE_1$  and  $b_2$  is assigned to  $PE_2$ . Identity behaviors  $n$  and  $w$  are added along with rendezvous channel  $sync$  to preserve the original control flow. Identity behaviors are represented by white boxes, while non-identity leaf behaviors are shown by shaded boxes.

The model representing the architecture is a parallel composition of hierarchical PE behaviors. Each PE behavior has, under its hierarchy, leaf level behaviors from the specification that were mapped to the corresponding PE. Therefore, by looking at the model on the RHS, we can identify that the architecture has two PEs and that  $b_1$  executes on  $PE_1$  and  $b_2$  executes on  $PE_2$ . Hence, the model reflects the system architecture and the mapping decision.

#### 4 Functional Equivalence Checking with BCGs

Once the BCG for a model is constructed, we apply the reduction rules (of Section 2.4) on it until the BCG can no longer be reduced. Thus, the normal form BCG of the model is obtained. If the normal form BCGs for two models are identical, then we can claim the models to be functionally equivalent. This is because the reduction rules preserve the functionality of the model. Thus, the equivalence test complies with our notion of functional equivalence.

Figure 8 shows how a BCG for the architecture model shown in RHS of Figure 7 is reduced to its normal form in a step wise fashion. At each step, we use the applicable reduction rule until we cannot apply them any more. The obtained normal BCG is identical to the BCG of the specification model, shown in the LHS of Figure 7. The topmost BCG in Figure 8 is derived from the architecture model. Note the VSP and VTP identity behaviors of the two PEs. Also, note the  $n$  and  $w$  behaviors added for synchronization. The channel  $sync$  in the architecture model is resolved using the principle shown in Figure 4. The resulting control dependencies are seen in the edges emanating from  $n$  and  $w$ .

The reduction to normal form takes place as follows. In the first step, we optimize away identity behaviors  $vsp_{pe1}$  and  $vsp_{pe2}$  using identity elimination rule R1. In Step 2, we use R1 again to optimize away node  $n$ . As a result, all edges emanating from  $n$  are replaced by those from  $b_1$ . Similarly, in step 3, node  $w$  is optimized away using R1. Hence, all edges from  $w$  are replaced by those from  $vsp_{m'}$ . Step 4 uses redundant control elimination rule R2 to get rid of redundant edges from  $vsp_{m'}$ . It may be noted that by definition of dominator in Section 2.4.2, we have  $vsp_{m'} \in dom(b_1, M')$ . Note that in the BCG after Step 3,

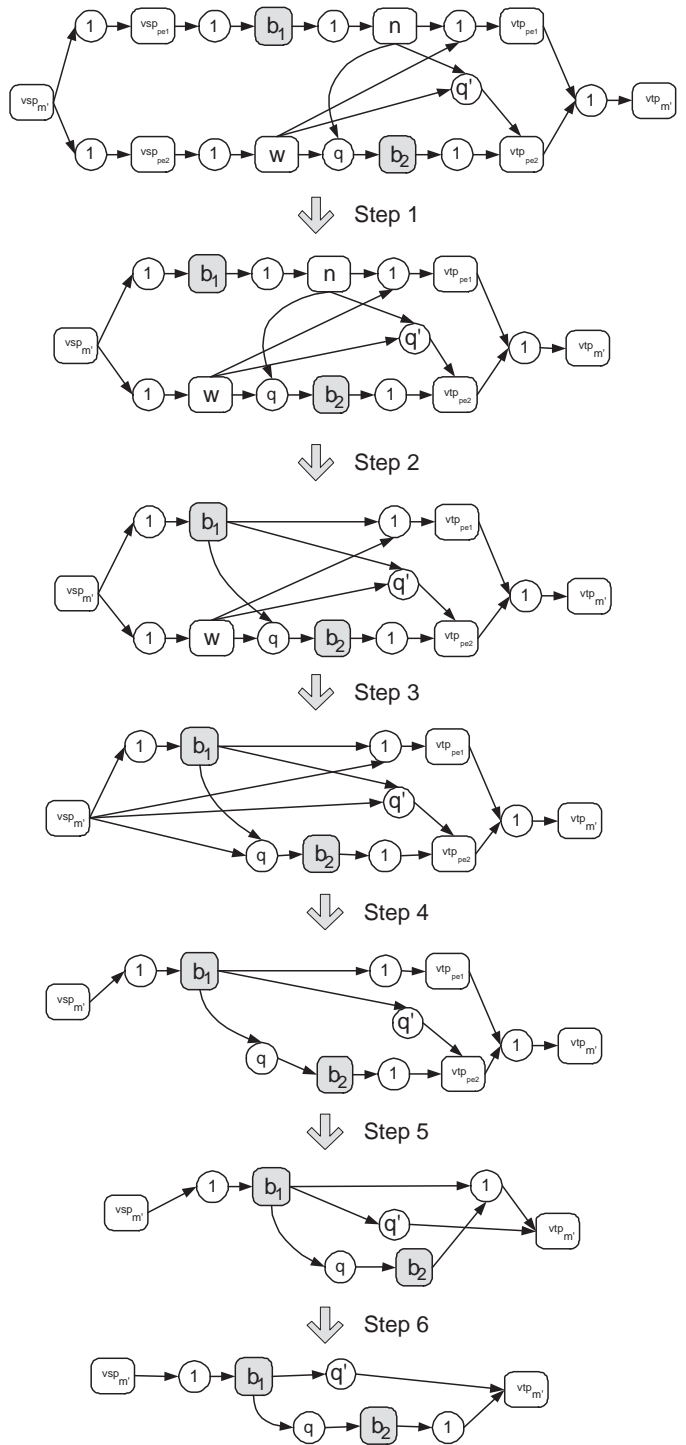


Figure 8. Reduction of architecture model BCG to specification model BCG

nodes labeled  $q$ ,  $q'$  and 1 have control dependencies from both  $b_1$  and  $vsp_{m'}$ . Since,  $vsp_{m'}$  is a dominator of  $b_1$ , we can eliminate the edges  $(vsp_{m'}, 1)$ ,  $(vsp_{m'}, q)$  and  $(vsp_{m'}, q')$ . In Step 5, we use R1 once again to eliminate identity behaviors  $vt_{ppe1}$  and  $vt_{ppe2}$ . Finally, in step 6, we use R2 to get rid of edge  $(b_1, 1)$ . This is possible because, from the definition of dominator, we can see that  $b_1 \in dom(b_2, M')$ . The BCG obtained after Step 6 cannot be reduced any further and is thus the normal form for the architecture model  $M'$ . It is also identical to the BCG of corresponding specification model, which is illustrated in Figure 3(a). Hence, we have shown the function equivalence of the two models before and after mapping of specification behaviors to PEs for our example.

Table 1. Equivalence checking for different mapping decisions

# PEs	# Channels	Extraction Time	Reduction Time
2 PEs	12	0.92s	3.13s
3 PEs	22	1.66s	4.73s
3 PEs	19	1.62s	4.28s
4 PEs	29	2.15s	6.55s
4 PEs	36	2.90s	9.02s

## 5 Experimental Results

An tool, consisting of two modules, was written in C++ for checking equivalence of specification and architecture level SpecC models. The BCG extractor module creates the BCG from a SpecC Model, while the BCG reducer module used the rules in Section 2.4 to generate the normal form BCG. An automatic architecture model generator [6] was used to create models for different architectural configurations and mapping decisions.

Experiments were done using a GSM voice codec application [4] used in cellular phones. The specification consisted of 43 leaf level behaviors (totaling over 11K lines of code) that were mapped to architectures containing 2 to 4 PEs. Different mapping decisions produced different number of synchronization channels as shown in Table 1. The BCG extraction and reduction times were in the order of a few seconds on a 2 GHz PC running RedHat Linux OS. The actual comparison of the normalized BCGs 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 mapping of behaviors in the specification to PEs in the architecture.

The main advantage of this techniques is that the architecture level model does not require any functional simulation. On the flip side, the equivalence checker needs to be aware of new behaviors and channels added during model transformation, so that it may resolve them into control dependencies. However, this is only to be expected since checking equivalence of two arbitrary models is intractable. In the future, we would like to extend the equivalence checking algorithm to validate more design steps like scheduling and communication synthesis.

## References

- [1] S. Abdi and D. Gajski. Automatic generation of equivalent architecture model from functional specification. In *Proceedings of the Design Automation Conference*, June 2004.
- [2] E. Barros and A. Sampaio. Towards provably correct hardware/software partitioning using occam. In *Proceedings of the International Workshop on Hardware-Software Codesign*, pages 210–217, June 2004.
- [3] D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, January 2000.
- [4] 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.
- [5] G. Kahn. The semantics of a simple language for parallel programming. In *Info. Proc.*, pages 471–475, August 1974.
- [6] J. Peng, S. Abdi, and D. Gajski. Automatic model refinement for fast architecture exploration. In *Proceedings of the Asia-Pacific Design Automation Conference*, pages 332–337, January 2002.
- [7] H. Saito, T. Ogawa, T. Sakunkonchak, M. Fujita, and T. Nanya. An equivalence checking methodology for hardware oriented c-based specifications. In *Proceedings of the High Level Design Validation and Test Workshop*, pages 274–277, October 2002.
- [8] T. Sakunkonchak and M. Fujita. Verification of synchronization in specc description with the use of difference decision diagrams. In *Proceedings of the Forum for Design Languages*, September 2002.