System Level Verification with Model Algebra

Samar Abdi and Daniel Gajski

Technical Report CECS-04-29
November 9, 2004

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{sabdi,gajski}@cecs.uci.edu

Abstract

This paper introduces Model Algebra (MA), a formalism for representing SoC designs at system level. We present the definition of Model Algebra and show how system level models can be represented as expressions in this formalism. We follow a system level design methodology, where design decisions gradually refine the functional specification model of the system to an architectural model with components and communication structure. The various design decisions are verified by checking the functional equivalence of models, before and after the design decision is implemented. For this purpose, we define the execution semantics and a notion of functional equivalence for system level models. Then, we present well defined rules for reducing a given model to a corresponding normal form. These rules are shown to be sound with respect to our notion of equivalence. We claim that two models are equivalent if they can be reduced to identical normal forms. As a result, it is possible to develop equivalence checkers to compare system level models for functional equivalence. Our approach promises significant savings in functional verification of system level models, because we require simulation for only the specification model. All models derived from the specification can be verified using equivalence checking and property verification.

Contents

List of Figures

System Level Verification with Model Algebra

Samar Abdi and Daniel Gajski
Center for Embedded Computer Systems
University of California, Irvine

Abstract

This paper introduces Model Algebra (MA), a formalism for representing SoC designs at system level. We present the definition of Model Algebra and show how system level models can be represented as expressions in this formalism. We follow a system level design methodology, where design decisions gradually refine the functional specification model of the system to an architectural model with components and communication structure. The various design decisions are verified by checking the functional equivalence of models, before and after the design decision is implemented. For this purpose, we define the execution semantics and a notion of functional equivalence for system level models. Then, we present well defined rules for reducing a given model to its corresponding normal form. These rules are shown to be sound with respect to our notion of equivalence. We claim that two models are equivalent if they can be reduced to identical normal forms. As a result, it is possible to develop equivalence checkers to compare system level models for functional equivalence. Our approach promises significant savings in functional verification of system level models, because we require simulation for only the specification models. All models derived from the specification can be verified using equivalence checking and property verification.

1 Introduction

The continuous increase in size and verification complexity of SoC designs has raised the abstraction level of system modeling. Since these abstract models are also simpler to understand and debug, the designer can hope to eliminate most functional errors early in the design process. Once the abstract system model is verified, it can be used as a source for deriving more detailed lower level models. As design decisions are made, the source model is refined to reflect those decisions. During design space exploration, the designer might need to create several refined models to represent the various design points. An important concern in such a design methodology is that the designer should not have to repeat costly simulations for each of the refined models.

The verification effort for the specification model must therefore be leveraged for verifying the refined models. An analogy can be seen in logic synthesis, where expensive gate level simulation is avoided by using logic equivalence checking. The RTL model, which simulates much faster than a gate level model, is verified as exhaustively as possible and then synthesized to a gate level implementation. The gate level and RTL models are then compared for equivalence using formal methods.

In recent years, not only are the RTL models increasing in size, a significant part of the design is being implemented in software. Hence, exhaustive simulation and debugging at the cycle accurate level is also becoming very time consuming. In an ideal scenario, one should need to simulate and debug only the abstract system specification model. Lower level models, that are derived from the specification, may be compared against the specification model using some formalism.

In this paper, we introduce **Model Algebra** (MA), which is a formalism for representing system level models and verifying their transformations. System level models, written in SLDLs, can be abstracted into MA expressions. Model transformations are realized by manipulation of their MA expressions. The formalism provides a set of reduction rules that can be used to derive a normal form expression for any given MA expression. Thus two models can be checked for equivalence by comparing their respective normal MA expressions.

The rest of the paper is organized as follows. In Section 2, we give an overview of our system level design and verification methodology. Section 3 discusses the requirements for modeling at the system level. In Section 4, we present the definition of Model Algebra in terms of its objects and composition rules. Construction of models with objects and composition rules of MA is discussed in Section 5. In Section 6, we deal with semantics of hierarchy and the impact of granularity on model analysis. The semantics of communication channels in MA is defined in Section 7. The formal execution semantics of models in MA is defined in

Section 8 and the resulting functional equivalence verification of MA models is discussed in Section 9. In Section 10, we look at each design step in our system level methodology and propose appropriate verification methods for them using MA. Based on our methods for comparing MA models, we present a case study in Section 11 for verifying the equivalence of input and output models for the first design step. Finally, we give a brief overview of related work in Section 12 and wind up with conclusions.

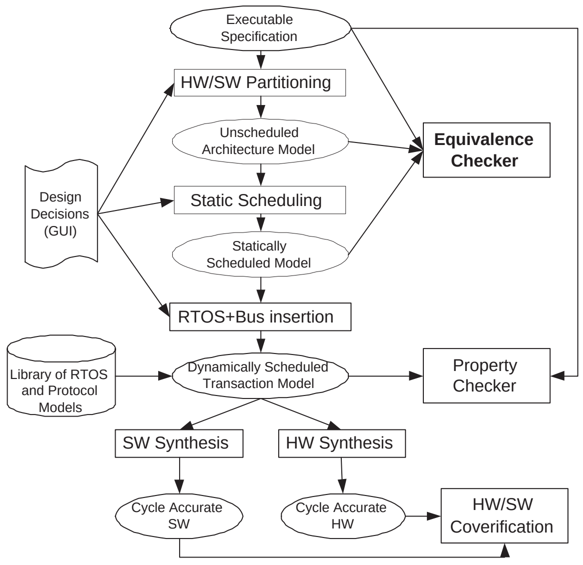


Figure 1. A possible system level design methodology

2 System Level Design and Verification Methodology

A possible system level design methodology is illustrated in Figure 1. We start by distributing the behaviors in the specification 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. During RTOS insertion, a scheduling policy is implemented for dynamic scheduling of behaviors mapped to SW. Bus insertion adds protocol 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.

A verification methodology is also shown along side the design methodology. From the above discussion, it can be seen that the steps of HW/SW partitioning and static scheduling transform the model by creating additional hierarchy of behaviors and creating a new schedule of their execution. During these steps, no new functionality is added. For this reason, we can employ an equivalence checker to verify the models resulting from such transformations. This reasoning shall be elucidated further when we look at examples of such transformations. Transformations resulting from steps such as RTOS insertion or communication synthesis involve the addition of new functionality. For instance, the RTOS is a new behavior that dynamically orders the execution of concurrent behaviors. Similarly, communication synthesis requires the addition of a new protocol and arbitration amongst transactions. These new behaviors cannot be comprehended by an equivalence checker. Therefore, to guarantee functional correctness of such transformations, we need to employ property checking methods like model checking or theorem proving. The property checker verifies that the functionality added by the new behaviors does not change the execution result of the model. Finally, cycle accurate models resulting from SW and HW synthesis may be verified using traditional co-verification tools.

3 Modeling Elements

A modeling formalism may be defined as a set of objects and composition rules that represent relationships between the objects. Our goal is to have a formalism that can allow the designer to express executable system models at different levels of abstraction. For instance, one should be able to express a model that shows only the functionality of the system using the objects and composition rules of the formalism. Also, one should be able to express models with structural details, using the same objects and composition rules. Given a model and its abstraction level, one should be able to identify the various structural artifacts within the model. Finally, a model expressed in such a formalism, should be executable so that it may be used to evaluate the design. The formalism must, therefore, have clear execution semantics.

A system can be viewed as a block of computation; with inputs and outputs for stimuli and response, respectively. This computation block is composed of smaller computation blocks that execute in a given order and communicate amongst themselves. Thus, for modeling purposes, it is imperative to have primitives for computation and communication. We will refer to the computation units as behaviors. A behavior has ports that allow it to be connected to other behaviors. The units of communication are variables and channels. These communication objects have different semantics. Variables allow a “read, compute and store” style of communication, while channels support a synchronized double handshake style of communication. Composition rules are used to create an execution order of behaviors and to bind their ports to either variables or channels. A system is thus represented as a hierarchical behavior composed of sub-behaviors communicating via variables and channels.

4 Model Algebra

An algebra may be defined as set of objects and relations amongst those objects, often referred to as signature of the algebra. The objects of MA can be defined as the tuple
$< \mathcal{B}, \mathcal{C}, \mathcal{I}, \mathcal{V} >$, where
$\mathcal{B}$ is the set of behaviors
$\mathcal{C}$ is the set of channels
$\mathcal{I}$ is the behavior interface
$\mathcal{V}$ is the set of variables

We also define a subset $\mathcal{B}^I$ of $\mathcal{B}$ representing the set of identity behaviors. Identity behaviors are those behaviors that, upon execution, produce an output that is identical to their input. In general, we will use the convention of naming identity behaviors as e followed by a subscript.

Each of the variables in $\mathcal{V}$ has *type* associated with it. The base type in MA is the *bit_vector*. All abstract data types like integer, float, boolean and even user defined structures and arrays can be converted to a bit vector representation. This representation, essentially, comes from the data organization in the memory of the PE. We define the subset $\mathcal{Q}$ to be the subset of $\mathcal{V}$ such that all data in $\mathcal{Q}$ is of type **boolean**.

4.1 Ports

Each behavior has an associated object called its interface. The interface carries the control and data ports of the behavior. In the case of a hierarchical behavior, the ports are by association of a variable to the interface. Hence, to internal behaviors of a hierarchical behavior, the port is seen as $\mathcal{I} < p >$, where $p \in \mathcal{V}$. The port is treated like any other local variable except that we allow only one kind of i/o operation on it. Local behaviors can either write to a port, in which case it is known as the *out-port*, or they may read from the port, in which case it is called the *in-port*. Ports of the *in-out* type are not allowed in MA. When the same port p is accessed from outside the behavior, it is identified by its association to the behavior. For example, in our case, port p of behavior b would be written as $b < p >$, as seen by external behaviors.

4.2 Addressing

Behaviors communicate with each other using either memory or channels. Essentially, memory based communication follows the SW programming paradigm, where one behavior writes data into a variable through an out-port and another behavior reads it via an in-port. Behaviors executing concurrently use synchronized data transactions amongst themselves for communication. Channels serve as the media for such transactions. Each transaction uses an address to identify the sender and the receiver behaviors. The transactions can, thus, be visualized to take place over virtual links, that are labeled by distinct addresses. Each of the links is associated with a channel. Hence, such a link may be identified as $c < a >$, where the link uses channel c and has the address a. Two transactions on a channel cannot share a link if they might take place simultaneously. In other words, all transactions on a single link must be totally ordered in time.

4.3 Composition Rules

Composition rules on the objects in MA are defined as relations in MA. These relations may contain two or more objects. Each composition rule creates a term, which may be further composed to create hierarchical behaviors.

4.3.1 Control flow

A control flow composition (R_c)determines the execution order of behaviors during model simulation. We write the relation as

$$q : b_1 \& b_2 \& ... \& b_n \rightsquigarrow b$$

where $\forall i, 1 \leq i \leq n b, b_i \in \mathcal{B} \cup \mathcal{I}, q \in \mathcal{Q}$. The composition rule implies that b executes after **all** the behaviors b_1 through b_n, called predecessors in the relation, have completed **and** q evaluates to TRUE. R_c is said to *lead to* b under the condition q. It implies a synchronization where b must wait for all predecessors. The degenerate case of the control flow relation is of the form $q_1 : b_1 \rightsquigarrow b$. Here, we only have a single predecessor, so b may start executing after b_1 if q_1 evaluates to TRUE, even if there are other control flow relations leading to b. If there are independent terms leading to b, they represent *program state machine* style transitions [5].

4.3.2 Non-blocking write

This composition rule (R_{nw}) is used to indicate that a behavior writes to a variable or an out-port of its parent behavior. In the case of a write to a data variable, we use the expression

$$b < p > \rightarrow v$$

where $b < p >$ is the out-port of the writing behavior and v indicates the memory into which the data is written. In its other manifestation, this composition rule can be used to create a port connection, written as

$$b < p > \rightarrow \mathcal{I} < p' >$$

In this case, the composition rule does not include any memory, but only indicates a port-map in a hierarchical behavior. Note that $< p' >$ must also be an out-port.

4.3.3 Non-blocking read

This composition rule (R_{nr}) is used to indicate that a behavior reads data from a variable or through an in-port of its parent behavior. In the case of a read from a data variable, we use the expression

$$v \rightarrow b < p >$$

where $b < p >$ is the in-port of the reading behavior and v indicates the memory from which the data is read. In its other manifestation, this composition rule can be used to create a port connection, written as

$$I < p' > \rightarrow b < p >$$

In this case, the composition rule does not include any memory, but only indicates a port-map in a hierarchical behavior. Note that $< p' >$ must also be an in-port.

4.3.4 Channel transaction

This composition rule (R_t) indicates a data transfer link from the sender behavior to one or more receiver behavior(s) over a channel. The semantics of the composition rule ensure that the sender and the receiver(s) are ready at the time of the transaction. In other words, it follows a rendezvous communication mechanism. The sender and receiver ports as well as the logical link of the channel are also indicated in the relation. We write this relation as

$$c < a >: b < p > \mapsto b_1 < p_1 > \& b_2 < p_2 > ... \& b_n < p_n >$$

where $b < p >$ is the out-port of the sending behavior and $b_1 < p_1 >$ through $b_n < p_n >$ are the in-ports of the receiving behaviors. The transaction takes place over channel c and uses the link addressed by $c < a >$.

4.3.5 Blocking write

This composition rule (R_{bw}) is used to indicate the port connection for the sender part of a transaction. The sender behavior writes to the out-port of its parent behavior through one of its own out-ports. Eventually, the port will be bound to a channel transaction. Thus, the blocking write relation facilitates the creation of hierarchy in the model. We represent a blocking write by the expression

$$b < p > \mapsto I < p' >$$

where $b < p >$ is the out-port of the writing behavior. The port $I < p' >$ on the parent behavior of b will eventually be bound to another blocking write relation or a channel transaction relation.

4.3.6 Blocking read

This composition rule (R_{br}) is used to indicate the port connection for the receiver part of a transaction link. The receiving behavior(s) read(s) from the in-port of their parent behavior through one of their own in-ports. Eventually, the port of the parent behavior will be bound to a channel transaction. Thus, the blocking read relation facilitates the creation of hierarchy in the model. We represent a blocking read by the expression

$$< a >: I < p' > \mapsto b_1 < p_1 > \& b_2 < p_2 > ... \& b_n < p_n >$$

where $b_1 < p_1 >$ through $b_n < p_n >$ are the in-port(s) of the receiving behavior(s). The port $I < p' >$ will eventually be bound to another blocking read relation or a channel transaction relation. The address of the virtual link ($< a >$) will be used for binding this port.

4.3.7 Grouping

This composition rule (R_g)is used to indicate a collection of composition rules. Essentially, grouping is used to create hierarchy of behaviors, by collecting the various compositions of sub-behaviors, local channels and local variables. This commutative relation is written as

$$r_1.r_2....r_n$$

where $\forall i, 1 \leq i \leq n$
$r_i \in \bigcup\{R_c, R_{nw}, R_{nr}, R_t, R_{bw}, R_{br}, R_g\}$.

4.4 Visualization of Objects and Composition Rules

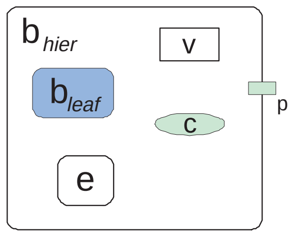


Figure 2. Visualization of various objects in MA

Figure 2 shows how we visualize the various objects of MA. A behavior is represented by a rounded rectangle, while a channel is represented by an ellipse. Variables inside behaviors are represented by rectangular boxes. Note that hierarchical behaviors like b_{hier} are shown by white rounded rectangles. Leaf level behaviors like b_{leaf}, that cannot be decomposed any further, are represented using colored rounded boxes. Identity behaviors like e are also shown as white rounded boxes. Ports are represented by little rectangles on the circumference of the box for corresponding behavior. A port labeled p may be seen for behavior b_{hier} in Figure 2.

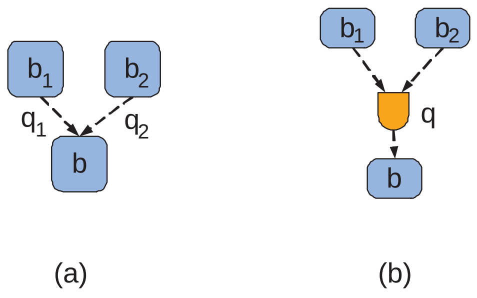


Figure 3. Visualization of control flow relations in MA

4.4.1 Control flow

Control flow relations are represented using broken directed edges as shown in Figure 3. A FSM-like control flow can be realized as shown in Figure 3(a). In this case, behavior b can start executing if **either** of the following conditions hold true:

1. b_1 has completed **and** q_1 evaluates to TRUE, **OR**
2. b_2 has completed **and** q_2 evaluates to TRUE.

Such a control flow can be expressed in MA as a grouping of the two control terms as follows

$$q_1 : b_1 \rightsquigarrow b.q_2 : b_2 \rightsquigarrow b$$

A more complex control flow is realized by the generic control relation that involves synchronization. This case is illustrated in Figure 3(b). The AND-gate symbol is used to indicate the synchronization before behavior b can start executing. In other words, b may start executing only if **both** b_1 and b_2 have completed and q evaluates to TRUE. This instance of control flow can be expressed with a single term as follows

$$q : b_1 \& b_2 \rightsquigarrow b$$

4.4.2 Data flow

Non-blocking communication takes place between behaviors using composition rules R_{nw} and R_{nr}. Essentially, behaviors read or write data to variables through their ports. The type of the port used and the variable should be the same for the relation to be valid. Figure 4(a) illustrates the non-blocking data flow from b_1 to b_2 via variable v. Behavior b_1 uses its out-port p_1 to write data to v, while b_2 uses its in-port p_2 to read data from v.

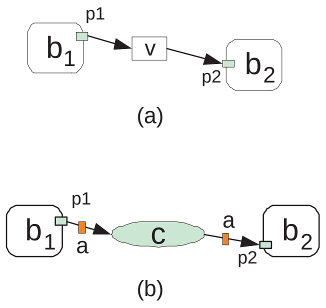


Figure 4. Visualization of data flow via ports

The channel transaction relation is illustrated in Figure 4(b). For now, let us consider only the simplest case of a channel transaction, that is a point-to-point transaction. Here, the port p_1 of b_1 uses the link a of channel c to write data. On the other side, port p_2 of b_2 uses the same link to read the data. The communication follows a double handshake protocol. The protocol guarantees that the receiver will wait until the sender is ready to write data. The sender on the other hand, will write data only upon the ready notification from the receiver. Hence, the channel semantics ensure that both the sender and receiver are synchronized at the time of the transaction.

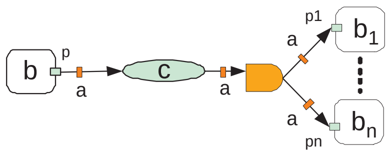


Figure 5. Visualization of multi-cast channel transaction

The more complex case of multi-cast channel transaction is shown in figure 5. The transaction consists of simultaneously sending the same data from a single sender to several receivers. For this reason, all the receiving behaviors and the sender must be executing concurrently. Also, a single address is used for a multi-cast transaction. The transaction link is visualized using a channel and the AND-gate symbol as shown in figure 5. The multi-cast communication still follows rendezvous semantics like the point-to-point communication. The difference is that instead of synchronizing two behaviors, all $n+1$ participating behaviors must be synchronized. The transaction link as shown in figure 5 can be expressed in MA as a single term

$$c < a >: b < p > \mapsto b_1 < p_1 > \& b_2 < p_2 > ... \& b_n < p_n >$$

5 Model Construction with MA

So far, we have seen the various objects and composition rules of MA. In this section, we look at how to construct hierarchical system models in MA. The objective is to represent models written in typical SLDLs using the objects and composition rules of MA. For simplicity, we will be using visual illustrations introduced in Section 4.4.

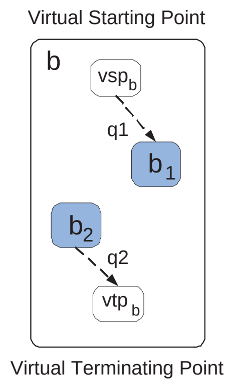


Figure 6. Control flow within hierarchical behaviors

5.1 Hierarchy

Using the control flow relations, we can compose behaviors such that they execute in a desirable order. Most SLDLs provide for hierarchical compositions of behaviors to aid modeling. In MA, hierarchy is achieved using the interface object and its relation to behaviors. Figure 6 shows a hierarchical behavior b consisting of sub-behaviors b_1 and b_2. The interface of b is visualized as the circumference of the box representing b. Note the *virtual starting point*(VSP) and the *virtual terminating point*(VTP) behaviors of b. The VSP is the identity behavior vsp_b that is the first to execute inside b. Other sub-behaviors of b are executed after vsp_b, depending on outgoing control relations from vsp_b. We can see in figure 6 that the VSP in this case vsp_b is triggering the execution of sub-behavior b_1. Due to its nature, a VSP behavior would only have outgoing control edges to other sub-behaviors of b. Similarly, the identity behavior vtp_b is the last behavior to execute inside b. In other words, the completion of b is indicated by the execution of vtp_b. Due to its nature, the VTP behavior will only have incoming edges from other sub-behaviors of b. All hierarchical behaviors are assumed to have a unique VSP and a VTP. Hence, the starting and terminating control relations of b can be written as

$$vsp_b \rightsquigarrow b_1.b_2 \rightsquigarrow vtp_b$$

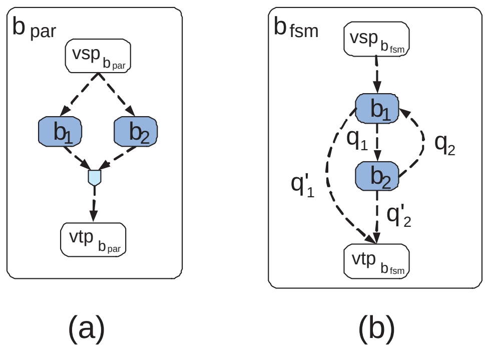


Figure 7. (a)Parallel and (b)FSM style compositions of behaviors

5.2 Parallel and Conditional Execution

Most SLDLs provide for special language constructs to create different types of behavioral hierarchies. The common ones are parallel composition and fsm-style composition. A sequential composition is simply a degenerate form of the fsm-composition. In MA, we can realize both these types of composition by using hierarchy and control relations.

Figure 7(a) shows a parallel composition of behaviors b_1 and b_2. A typical SLDL may allow construction of a parallel composition using a statement like
par $\{$**run** b_1; **run** $b_2\}$.
Let the resulting behavior be called b_{par}. The execution of b_{par} indicates that both b_1 and b_2 are ready to execute. The execution of b_{par} completes when both b_1 and b_2 have completed. In the corresponding MA expression, $vsp_{b_{par}}$ and $vtp_{b_{par}}$ serve as the starting and terminating points, respectively, of the hierarchical behavior b_{par}. We can see, that inside b_{par}, b_1 and b_2 are allowed to start simultaneously. This is ensured by the control relations
$vsp_{b_{par}} \rightsquigarrow b_1.vsp_{b_{par}} \rightsquigarrow b_2$
Hence, the parallelism is realized by orthogonality of the execution of behaviors b_1 and b_2. The control relation at the end
$b_1 \& b_2 \rightsquigarrow vtp_{b_{par}}$
ensures that both b_1 and b_2 must complete their execution before $vtp_{b_{par}}$ executes. The execution of $vtp_{b_{par}}$ indicates the completion of the hierarchical behavior b_{par}.

A typical FSM style composition of behaviors is shown in Figure 7(b). The control flow between behaviors is typically expressed using switch-case or goto constructs in SLDLs. A simple pseudo code example for a hierarchical behavior b_{fsm} is as follows

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

The control relations of b_{fsm} can be written as follows in

MA
$vsp_{b_{par}} \rightsquigarrow b_1.q_1 : b_1 \rightsquigarrow b_2.q'_1 : b_1 \rightsquigarrow vtp_{b_{par}}$.
$q_2 : b_2 \rightsquigarrow b_1.q'_2 : b_2 \rightsquigarrow vtp_{b_{par}}$

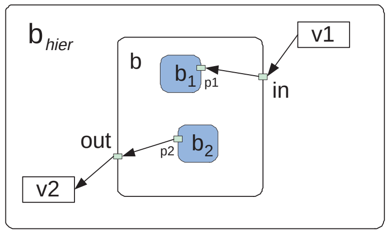


Figure 8. Using ports for non-blocking data flow in hierarchical behaviors

5.3 Variable Access via Ports

In MA, as in most SLDLs, a variable is directly visible only to the behaviors that are at the same level of hierarchy as the variable itself. Therefore, in order to access variables at higher levels of hierarchy, data ports are used. As shown in Figure 8, behavior b_1 reads variable v_1 present in b_{hier} via the port "in" of its parent b. Hence, to realized this port connection, we need terms at different levels of behavior hierarchy. At the level of b_{hier}, we use the non-blocking relation

$$v_1 \rightarrow b < in >$$

At the level of b, we use the port connection from the interface of b to b_1. We can write this as the relation

$$I < in > \rightarrow b_1 < p_1 >$$

The dual of read port connection is the write port connection as shown by the access of variable v_2 from behavior b_2 in figure 8. In this case, the port "out" of b is used to realize the variable access. The term at the level of b_{hier} is

$$b < out > \rightarrow v_2$$

while the term at the level of b is

$$b_2 < p_2 > \rightarrow I < out >$$

5.4 Channel Access via Ports

Non-blocking communication is typically used for sequentially executing behaviors. The sender behavior writes to the communicating variable. The receiver behavior executes after the writer has completed and reads from the communicating variable. However, when behaviors are executing concurrently, such a method of communication would not be safe anymore. In other words, we cannot guarantee the actual order of execution of concurrent behaviors and hence, it is not possible to tell if the receiving behavior will execute after the sender. To allow safe and predictable data transfer between behaviors, we use a channel transaction.

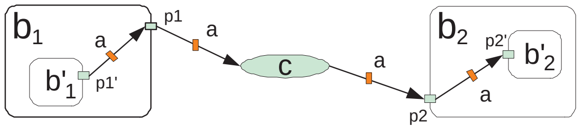


Figure 9. Blocking data flow bound to channel

As in the case of non-blocking reads and write, MA provides mechanism for blocking reads and writes via ports. For instance, in Figure 9, we see a channel transaction from b_1 to b_2 over c. After zooming into the hierarchy of b_1 and b_2, we see that the transaction is taking place from b'_1 to b'_2. The port p_1 of b_1 makes the channel c visible to b'_1. Therefore, using the relation

$$< a >: b'_1 < p'_1 > \mapsto I < p_1 >$$

behavior b'_1 can access channel c. However, this requires p_1 to be bound to the virtual link addressed by a. Similarly, on the other side, sub-behavior b'_2 inside b_2, uses the blocking relation

$$< a >: I < p_2 > \mapsto b'_2 < p'_2 >$$

to access the read method of c via port p_2. In this case, port p_2 makes the channel c visible to b'_2. As before, p_2 must be bound to the virtual link addressed by a.

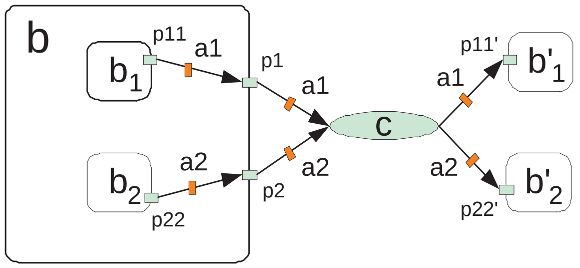


Figure 10. Sharing channel for transactions with different addresses

In MA several virtual links may share a single channel. Each of the virtual links are assigned a different address, but the data transfer takes place on the same medium. Figure 10 shows an instance of channel sharing. In this model, we have two virtual links with addresses $< a_1 >$ and $< a_2 >$. Transactions may be attempted concurrently on these links. However, due to sharing of the channel, we can allow only one transaction at a time. This is a classic case of bus arbitration, where an arbiter ensures that only one transaction

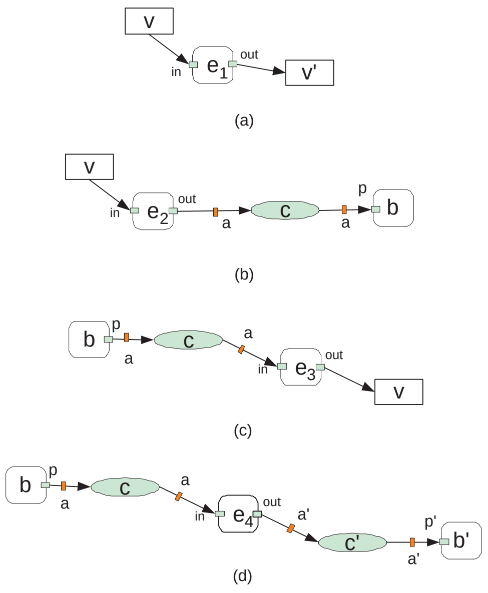


Figure 11. Various manifestations of the identity behavior

may take place over a bus at any time. In MA, the same concept is implemented using mutual exclusion in the channel. The read and write methods of the channel implement a mutual exclusion policy, where the channel is a shared resource and each transaction is treated as a critical section. This allows us to connect several different virtual links to the same channel.

5.5 Using Identity Behaviors

A class of behaviors in MA is known as the identity behavior. As the name suggests, these behaviors have the same output as the input. As a result they do not have any computation inside them. They have two ports namely the "in" port for reading the input and an "out" port for writing the output. In general, the identity behavior first reads data from the "in" port to a local variable and then writes this variable to the "out" port. The actual implementation of the read and write within the identity behavior depends on the port connections.

There are four basic manifestations of the identity behavior as shown in figure 11. Let us assume that the data read and written by the identity behavior is of integer type. In the first case, as shown in figure 11(a), both the "in" and "out" ports of the identity behavior e_1 are connected to variables. Hence, the respective read and write are non-blocking relations. In MA, the read/write relations of e_1 are expressed as

$$v \rightarrow e_1 < in > .e_1 < out > \rightarrow v'$$

The typical SLDL implementation of e_1 would look like

```
behavior e1 (in, out){
        int temp;
        temp = in;
        out = temp;
};
```

The second case of identity behavior is shown in figure 11(b). Here, the "in" port is connected to a variable, hence the input is read using a non-blocking relation. On the other hand, the "out" port is connected to channel c. Hence, the output needs to be sent to b using a blocking write relation. In MA, the read/write relations of e_2 are expressed as

$$v \rightarrow e_2 < in > .c < a >: e_1 < out > \mapsto b < p >$$

The typical SLDL implementation of e_2 would be as follows

```
behavior e2 (in, out){
        int temp;
        temp = in;
        out.write(a, temp);
};
```

The third case of identity behavior is shown in figure 11(c). Here, the "in" port is connected to a channel c, hence the input is read from behavior b using a channel transaction. On the other hand, the "out" port is connected to variable v. Hence, the output needs to be written using a non-blocking write relation.In MA, the read/write relations of e_3 are expressed as

$$c < a >: b < p > \mapsto e_3 < in > .e_3 < out > \rightarrow v$$

The typical SLDL implementation of e_3 would be as follows

```
behavior e3 (in, out){
        int temp;
        in.read(a, &temp);
        out = temp;
};
```

Finally, the fourth manifestation of identity behavior is shown in figure 11(d). Here, the "in" port of e_4 is connected to a channel c for reading data from b. Hence the input is read using a channel transaction relation. The "out" port of e_4 is also connected to a channel named c' for writing data to b'. Hence, the output is also written using a channel transaction relation. In MA, the read/write relations of e_4 are expressed as

$$c < a >: b < p > \mapsto e_4 < in > .c' < a' >: e_4 < out > \mapsto b' < p' >$$

The typical SLDL implementation of e_4 would be as follows

```
behavior e4 (in, out){
        int temp;
        in.read(a, &temp);
        out.write(a', temp);
};
```

6 Hierarchical Modeling in MA

The model of a system is a behavior in MA. Typically, it is a hierarchical behavior showing the various components and connections of the system and the functionality within these components. While modeling, it is imperative to provide the right amount of detail for analysis purposes. The granularity of the leaf level behaviors is an important factor in deciding if the model can be analyzed. Typically, leaf behaviors are treated as atomic by the model analysis and transformation tools. In one extreme case, a system model can be represented as a single leaf behavior. Although the model may simulate correctly, it is useless for performing any transformations. On the other hand, too much granularity may make design decisions too cumbersome. For example, each statement in the SLDL description may be treated as a leaf behavior. Such a description will present too many design choices, with only few of them being useful. Usually, the designer knows what functionality should not be distributed on different PEs. For instance, operations that work on the same set of data or use the same type of resources are grouped into one behavior.

6.1 Internal and Interface Terms

As mentioned earlier, hierarchy is a key feature in SLDLs. Hierarchy allows us to compose systems in a modular way. In MA, it is possible to represent a behavior as a grouping of terms involving its sub-behaviors, its interface and its local variables and channels.

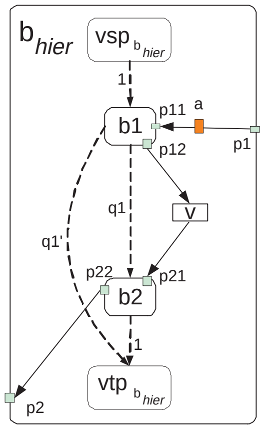


Figure 12. A hierarchical behavior with local objects and relations

Figure 12 shows a hierarchical behavior b_{hier}. The expression for the hierarchical behavior is written using the local objects and composition rule. For instance, in the given behavior b_{hier}, we can see sub-behaviors b_1 and b_2. We can also see control flow relations that determine the execution scenario under the conditions labeled on the control arcs. We also see data flow relations, both amongst sub-behaviors and between sub-behaviors and the interface. The grouping of relations between local objects will be referred to as the *internal terms* of a hierarchical behavior. Similarly, the grouping of relations involving the interface will be referred to as the *interface terms* of the hierarchical behavior.

We can write the hierarchical behavior as a grouping of all its internal and interface terms, along with the internal terms of its sub-behaviors. The grouping of internal terms for a given behavior b is represented as $[b]$. Thus, we can write

$[b_{hier}] = [vsp_{b_{hier}}].[b_1].[b_2].[vtp_{b_{hier}}].vsp_{b_{hier}} \rightsquigarrow b_1.$
$q_1 : b_1 \rightsquigarrow b_2.q_1' : b_1 \rightsquigarrow vtp_{b_{hier}}.b_2 \rightsquigarrow vtp_{b_{hier}}.b_1 < p_{12} > \rightarrow v.$
$v \rightarrow b_2 < p_{21} >$

The interface terms of b_{hier} is represented by $|b_{hier}|$. From figure 12, we can see that

$|b_{hier}| =< a >: I < p_1 > \mapsto b_1 < p_{11} > .$
$b_2 < p_{22} > \rightarrow I < p_2 >$

Finally, we write the hierarchical behavior as a grouping of its internal and interface terms. We will use the convention of enclosing the expression for a hierarchical behavior in braces. Therefore, we get

$b_{hier} = ([b_{hier}].|b_{hier}|)$

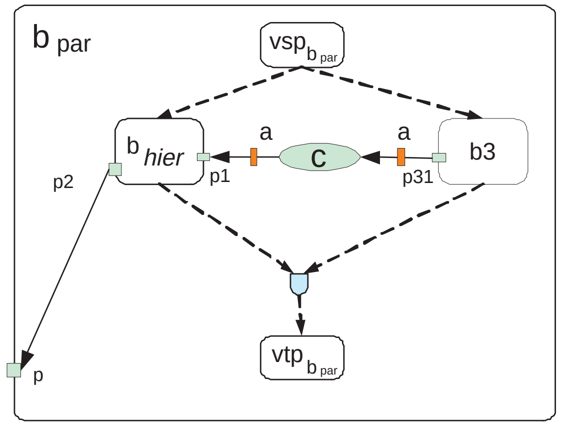


Figure 13. Hierarchical behavior b_{par} with a parallel composition

6.2 Multiple Levels of Hierarchy

In the above example, a *fsm*-like hierarchical composition was created. The resulting behavior b_{hier} can be used further to create more hierarchical behaviors. For instance, in figure 13, we see behavior b_{hier} in a parallel composition with behavior b_3. The two behaviors exchange data using

the virtual link addressed a, over channel c. The hierarchical composition results in a new behavior called b_{par}. The expression for b_{par} is written in MA as follows
$b_{par} = ([vsp_{b_{par}}].[b_{hier}].b_3.[vtpb_{par}].vsp_{b_{par}} \rightsquigarrow b_{hier}.$
$vsp_{b_{par}} \rightsquigarrow b_3.b_{hier}\&b_3 \rightsquigarrow vtpb_{par}.$
$c < a >: b_3 < p_{31} > \mapsto b_{hier} < p_1 > .$
$b_{hier} < p_2 > \rightarrow I < p >)$

6.3 Flattening of Hierarchical Behaviors

Hierarchy is only a modeling artifact in MA. Addition of hierarchy allows the designer to group different behaviors together. It does not add any functionality to the model. Unlike SLDLs, MA does not have different types of hierarchical compositions. Hierarchy by itself does not influence how a particular set of behaviors would execute. That execution order is already captured using the control flow and data transaction relations in MA. The usefulness of hierarchy comes in representing the structural entities in the model. For instance, in an architecture model, different PEs execute different sets of behaviors. These sets of behaviors are grouped into different hierarchical PE behaviors.

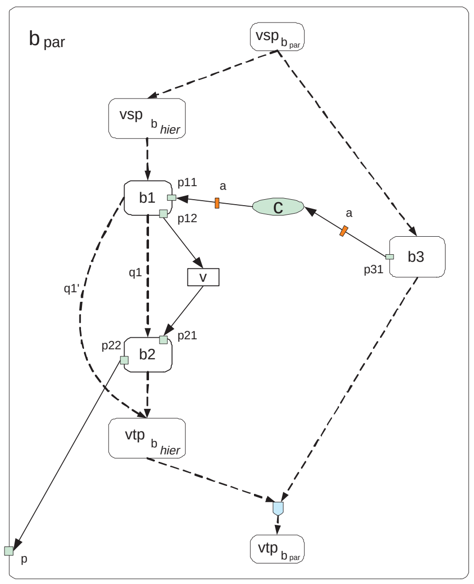


Figure 14. Behavior b_{par} after flattening of b_{hier}

For functional validation, we need to be concerned with only the leaf level behaviors. Hence, we may get rid of hierarchy by flattening the model. The laws for flattening a hierarchical behavior follow from the semantics of hierarchical behaviors in MA. Consider the hierarchical behavior b_{hier} in figure 12. Now, by the semantics of the VSP, any control relation leading to b_{hier} is effectively leading to $vsp_{b_{hier}}$. This is because $vsp_{b_{hier}}$ is always the first behavior to execute inside b_{hier}. Similarly, in any control relation where b_{hier} is a predecessor, it may be replaced by $vtp_{b_{hier}}$. This is because $vtp_{b_{hier}}$ is always the last behavior to execute inside b_{hier}.

This allows us to define the first two laws for flattening a given hierarchical behavior b. The term on the LHS is part of the original expression involving b. The term of the RHS is the one that replaces the LHS term once b is flattened. We will use symbols $x, yandz$ as free variables.

FL 1 $q : x \rightsquigarrow b \Longrightarrow q : x \rightsquigarrow vsp_b$

FL 2 $q : b \rightsquigarrow x \Longrightarrow q : vtp_b \rightsquigarrow x$

To enable data flow, hierarchical behaviors allow for ports on their interface. These ports are essentially a conduit for data transfer from one leaf behavior to another. During flattening, these ports can be optimized away by appropriately making new port connections as shown in figure 14. A virtual link addressed a over channel c is used for blocking data transfer from b_3 to b_1. However, due to the hierarchical behavior b_{hier}, channel c is not visible from the local scope of b_1. Thus, the port p_1 is used to facilitate the connection of b_1 with channel c. When the interface of b_{hier} disappears during flattening, we can directly connect channel c to b_1. Similarly, the port p_2 on b_{hier} can be optimized away by directly connecting $b_2 < p_{22} >$ to port p on b_{par} interface.

Therefore, we have the following additional laws for port optimization during behavior flattening. On the LHS, we show the expression for the hierarchical behavior enclosed in braces. Only the interface term for the relevant port is shown.

FL 3 $(...y \rightarrow I < p > ...) < p > \rightarrow x \Longrightarrow y \rightarrow x$

FL 4 $x \rightarrow (...I < p > \rightarrow y...) < p > \Longrightarrow x \rightarrow y >$

FL 5 $z < a >: x \mapsto (... < a >: I < p > \mapsto y...) < p > \Longrightarrow$
$z < a >: x \mapsto y >$

FL 6 $z < a >: (... < a >: y \mapsto I < p > ...) < p > \mapsto x \Longrightarrow$
$z < a >: y \mapsto x >$

6.4 Granularity of Leaf Behaviors

From the verification perspective, it is important that the model should have a fine enough granularity of leaf behaviors. Recall that during analysis of system models, we treat leaf behaviors as atomic units. Channel transactions, and consequently blocking reads and writes, impose implicit control relations between communicating behaviors. Therefore, such communication points must be explicitly represented in the model description.

Our goal is to clearly distinguish the control relation between two leaf behaviors. This relationship will form the basis for comparing two models for functional equivalence. If there is a leaf level behavior with a blocking read or write

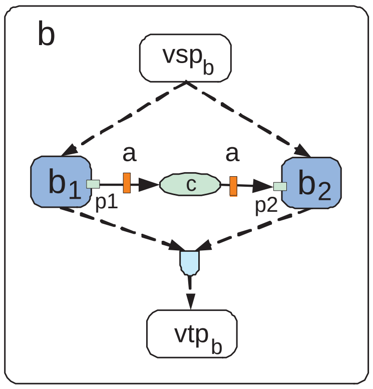


Figure 15. Leaf level behaviors communicating using channel c

to a channel, then during execution it must wait at some unknown point to complete that blocking transaction. The lack of granularity, therefore, restricts us from knowing the actual order of computations. This problem is illustrated by a simple example in Figure 15. Consider a hierarchical behavior b formed by the parallel composition of leaf behaviors b_1 and b_2. Behaviors b_1 and b_2 communicate in a rendezvous fashion, using channel c. Since leaf behaviors are atomic for our analysis, we cannot tell exactly when does the transaction over c take place. In other words, we cannot tell what part of b_1 or b_2 executes before the transaction and what part executes after it. By the execution semantics of the channel transaction, there is a control dependence between parts of b_1 and b_2. However, due to the lack of granularity in the description, we cannot determine this dependence.

The importance of granularity is further illustrated in Figure 16. This time we assume that behaviors b_1 and b_2, from Figure 15, are hierarchical. Thus, the MA description now provides more details, so that the control ordering imposed by the transaction relation can be analyzed. Both b_1 and b_2 are sequentially composed. The channel is linked to identity behaviors on either side, namely e_1 and e_2. By virtue of being identity behaviors, both e_1 and e_2 do not carry any computation. Therefore, for a channel transaction involving e_1 and e_2, we need not be concerned about any hidden ordering of computation. Essentially, by the rendezvous semantics of the transaction, all execution preceding e_1 also precedes all execution following e_2, and vice versa.

In the first scenario, shown in Figure 16(a), we see that b_{12} inside behavior b_1 executes before e_1. Considering the rendezvous semantics of the channel transaction, we can tell that b_{12} has no ordering with b_{21}. However, in the scenario shown in figure 16(b), the same rendezvous semantics force b_{21} to execute before b_{12}. This is because now, the control flow inside b_1 makes e_1 execute before b_{12}. Therefore. in the above two scenarios, the execution order of leaf behaviors are different. Note that if both behaviors b_1 and b_2 were treated as leaf behaviors, the two scenarios shown in figure 16 could **not** be distinguished.

Based on the above discussion, we can establish the simple rule for granularity of analyzable behaviors. We impose the modeling restriction that channel transaction relations or blocking data flow relations can involve only hierarchical behaviors or identity behaviors. Hence, after flattening the model may have channel transaction relations only between identity behaviors. In other words, in a completely flattened analyzable model, if there is a term $c < a >: x \mapsto y$, then $x, y \in B^{\varepsilon}$.

7 Channel Semantics

The channel object allows for reliable communication between two concurrently executing behaviors. In an SLDL implementation, the channel uses events and data variable to implement a rendezvous communication protocol. As discussed before, a channel transaction implies a control dependency between parts of the communicating behaviors. Hence, we will assume both the sender and the receiver to be identity behaviors in future discussions.

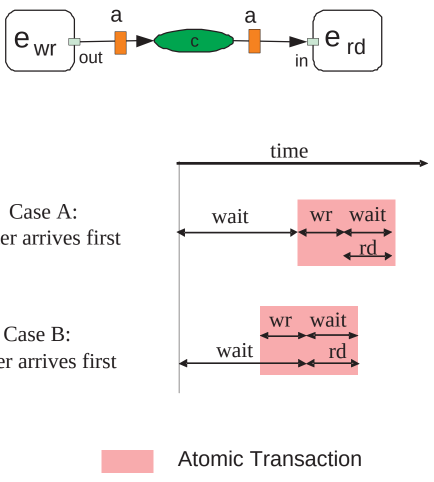


Figure 17. Timing diagram of a transaction on a channel

7.1 Channel with Single Virtual Link

Figures 17 shows a transaction taking place over channel c. We can express this transaction MA using the term

$$c < a >: e_{wr} < out > \mapsto e_{rd} < in >$$

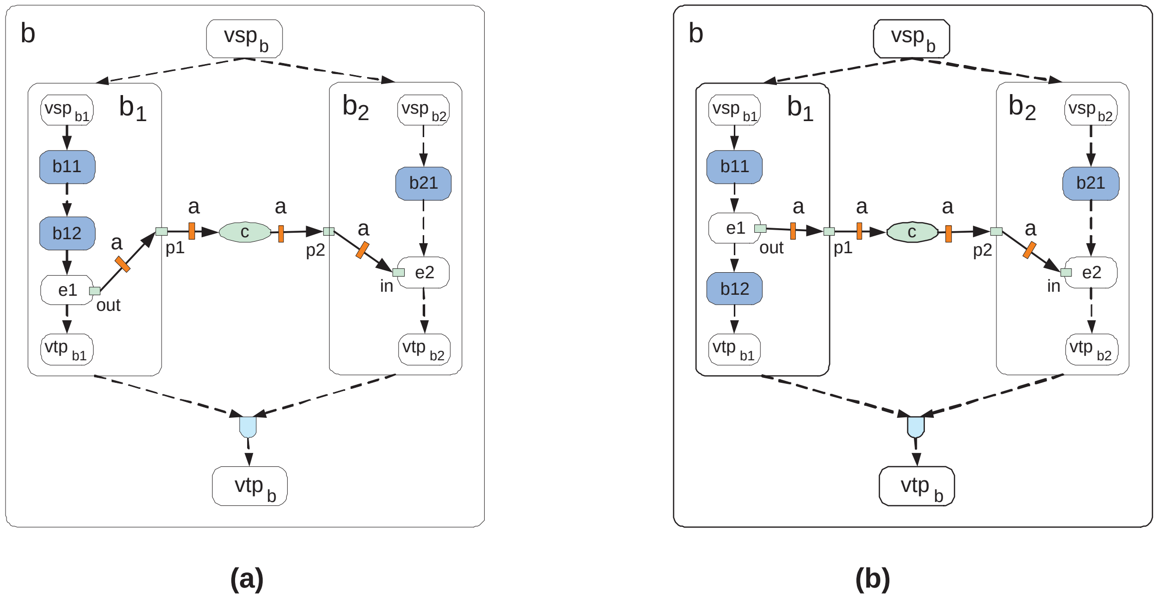


Figure 16. Different scenarios for transaction over c (a) b_{12} executing before e_1, and (b) b_{12} executing after e_1

The timing diagram for this channel transaction shows two instances of execution. In the first instance, called Case A, the writer reaches the communication point before the reader. By this we mean that during model execution, e_{wr} is scheduled to execute before e_{rd}. However, the rendezvous semantics dictate that e_{wr} must wait until e_{rd} is ready before executing. It may be noted that if there is a control dependency from e_{wr} to e_{rd}, the resulting model would deadlock. Hence, e_{rd} must be allowed to start independently of e_{wr} and vice versa. Once e_{rd} is ready to start the transaction, it notifies e_{wr}. The transaction is thus initiated by e_{wr}, that performs a write on the local memory of the channel. Subsequently, e_{rd} reads the data from this memory.

In the second execution scenario, called Case B, the reader is scheduled before the writer is ready. This forces e_{rd} to wait until e_{wr} is ready to start executing. The shaded part of the execution, in the timing diagram, indicates the atomic nature of the transaction. Note that the channel resources (i.e. its local memory) are occupied only during the actual reading and writing of the data.

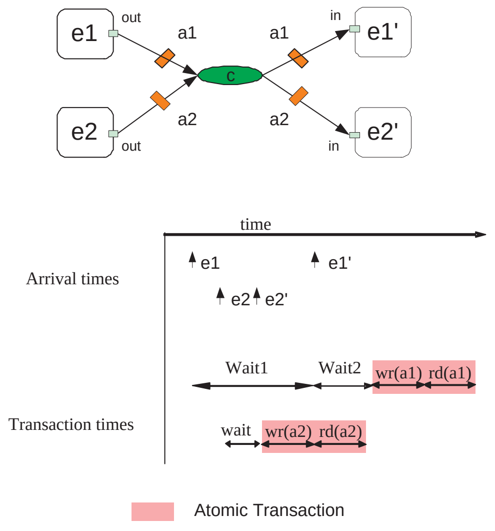


Figure 18. Multiple simultaneous transactions on a single channel

7.2 Channels with Multiple Virtual Links

As discussed earlier, channel sharing is possible for different virtual links, but the transactions are scheduled sequentially. This mutual exclusivity of transactions can be achieved by the use of semaphore (or test and set) constructs in SLDL. Thus, the shaded part representing the actual data read and write over the local memory of channel is mutually exclusive. The reason why we do not make the entire transaction (including synchronization) mutually exclusive is to allow greater bandwidth on the channel. Consider the configuration shown in figure 18. In this case, two virtual links, addressed a_1 and a_2, are shared over channel c. These links can be written as a grouping of the following terms
$c < a_1 >: e_1 < out > \mapsto e_1' < in >$.
$c < a_2 >: e_2 < out > \mapsto e_2' < in >$
The timing diagram shows the actual arrival schedule of the four communicating identity behaviors and the resulting communication schedule on the channel. Note that de-

spite the fact that e_1 arrives first, transaction on a_2 takes place before that on a_1. This is because, the data transfer of transaction on a_2 is ready to be performed before that for a_1. Thus, the data transfers on the channel are scheduled on first-ready first-serve basis. Although the transaction on a_1 is ready to be performed when e_1' arrives, it must wait for the duration $wait_2$ since the transaction on a_2 is in progress.

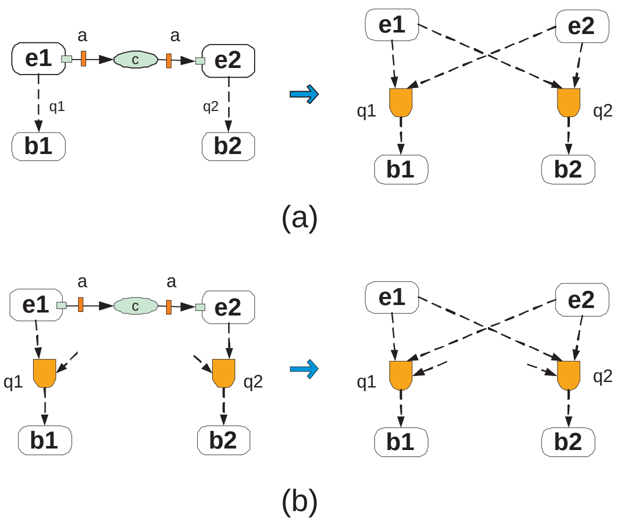


Figure 19. Resolution of channels into control dependencies

7.3 Control Flow Resolution of Links

As seen during the discussion of channel semantics, the channels in MA imply control flow dependencies between communicating behaviors. Our eventual goal is collect all control dependencies between behaviors to form a monolithic control flow graph of behaviors. We will now see how to resolve the virtual links in flattened MA models into control dependencies. Figure 19 demonstrates this control dependency extraction.

Recall that in an analyzable model, blocking relations and channel transaction relations can involve only identity behaviors or hierarchical behaviors. Upon flattening, the analyzable model would only have channel transaction relations between identity behaviors. Thus, for the purpose of control flow extraction from channel transaction relations, we need to consider only the case where sender and receiver are both identity behaviors.

The synchronization properties of a channel would ensure the following two premises:

1. Any behavior following the sender identity behavior would not execute until the receiver identity behavior has executed.
2. Any behavior following the receiver identity behavior would not execute until the sender identity behavior has executed.

If we were to optimize away the channel to extract only the control dependencies, the result will be as shown in figure 19. As per the above premises, behavior b_1 following sender e_1 cannot start until e_2 has completed. This is guaranteed by including the term

$$q_1 : e_1 \& e_2 \rightsquigarrow b_1$$

In the dual of the above case, b_2 following e_2 is blocked until the sender e_1 has executed. This premise is ensured by the term

$$q_2 : e_2 \& e_1 \rightsquigarrow b_2$$

Figure 19(b) shows the general case, where the behaviors following the sender and the receiver may already have several predecessors. In that case, the new predecessor (e_1 for b_2 and e_2 for b_1) is simply added to the list of predecessors in the corresponding blocking relation.

8 Execution Semantics

In order to define the execution semantics of MA, we first introduce the underlying model of computation. The control flow in the model is captured using the *Behavior control graph*(BCG). BCG is similar to the popular computation model of Kahn process network (KPN). KPN is a directed graph, where nodes represent processes and edges represent unbounded FIFO queues. Each edge is directed from the writer to the reader process. Also, writes are unblocked, while reads are blocking. This means that the reading process must wait until the queue the required amount of data for the reader process to execute. Note that all queues have only one reader and only one writer, and that the queues are the medium for data transfer between processes. KPN can effectively model concurrency and synchronization, but they are not useful for modeling non-determinate behavior or conditional control flow. The data flow in the model is captured using the *Port Connection Network* (PCN), which is a net-list of behaviors, variables and control conditions, with directed arcs denoting the data dependencies amongst them. Together, the BCG and the PCN are used to define the execution semantics of the model.

8.1 Behavior Control Graph

The BCG is similar in principle to the Kahn Process Network [10], 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

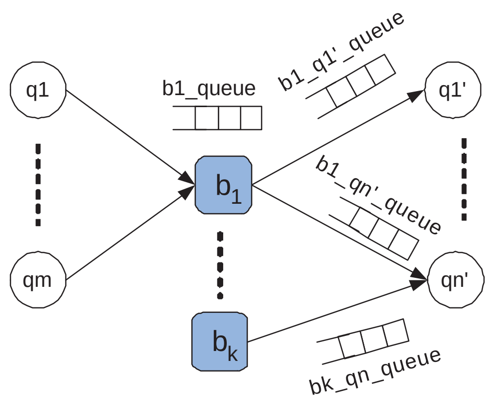


Figure 20. The firing semantics of BCG nodes

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 20. 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.

8.2 Deriving BCG from MA Expression

Now that we have described the BCG, we can create a unique BCG from a given MA expression. This will allow us to establish the execution semantics of MA. We have seen how to create a flattened behavior for a model in section 6.3. We also saw the control relations that may result from a communication channel. After, flattening the model and extracting the control dependencies, we are left with a set of leaf level behaviors and control relations amongst them. This can be directly translated into a BCG in a trivial fashion. For each leaf behavior, we introduce a behavior node in the BCG, labeled by the leaf behavior's id. For each control relation, we introduce a control node in the BCG, labeled by the control condition id. Also, edges are added to the BCG depending on the control relation. For instance, a control relation of the form

$$q : b_1 \rightsquigarrow b_2$$

would imply two directed edges (b_1, q) and (q, b_2) in the BCG. On the other hand, a control relation of the form

$$q : b_1 \& b_2 ... \& b_n \rightsquigarrow b$$

would imply $n+1$ directed edges (b_1, q), (b_2, q),...,(b_n, q) and (q, b) in the resulting BCG.

Recall that each hierarchical behavior has unique *vsp* and *vtp* identity behaviors. Let us assume that the top level behavior in the model is called m. Then vsp_m is the first node to fire in the BCG of m. Therefore, m may be simulated by placing a single token in the queue for vsp_m. The simulation terminates when there are no more tokens left to be consumed. In other words, if all the FIFOs in the BCG are empty, then the execution has terminated.

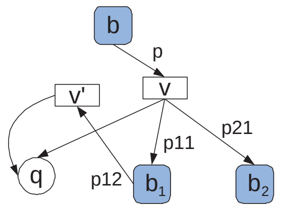


Figure 21. Port connection network showing data dependencies

8.3 Port Connection Network

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 21. 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.

8.4 Deriving PCN from MA Expression

In MA, a non-blocking write is represented by the relation

$$b < p > \rightarrow v$$

In a PCN, this results in a directed edge from a behavior node b to a variable node v would mean that b writes to the storage indicated by v. The edge label p indicates the out port used by b for writing v. Similarly, the non-blocking read relation

$$v' \rightarrow b' < p' >$$

results in an edge from a variable node v' to a behavior node b', labeled p', indicating that b' reads variable v' using its in port p'. We must note that for each variable, there can be only one writer behavior. The restriction of having a single writer behavior for each variable would simplify modeling and analysis, since we do not have to deal with hazards related with multiple writers.

Finally, the channels are also impose edges in the PCN. In the flattened form, we would expect to see channel relations only between identity behaviors. A channel transaction, represented by the MA relation

$$c < a >: e_1 < out > \mapsto e_2 < in >$$

will result in a directed edge from e_1 to e_2 in the PCN. A multi-cast transaction of the type

$$c < a >: e < out > \mapsto e_1 < in > \& e_2 < in > \& ... \& e_n < in >$$

will result in n edges in the PCN, each such edge originating at e and terminating at nodes e_1 through e_n.

9 Equivalence Checking of Models

The motivation behind MA is to enable the functional verification of various model transformations taking place during system level design. As a result of every design decision, the system model is transformed to reflect the properties imposed by the design decision. However, we must be able to ensure that the original intended functionality has not changed as a result of this transformation. We can ensure this **either** by using only proven correct transformations, **or** by having an equivalence checking tool to verify the models before and after the transformation. In either case, we need a notion of functional equivalence of models in MA. Using these notions, we can build tools for checking if two MA models are functionally equivalent. In this section, we will define our notion of equivalence and the algorithms needed for comparison of models based on such notion.

9.1 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
$\forall v, I(M) = I(M') \Rightarrow \tau(v, M, I(M)) = \tau(v, M', I(M'))$

From the above discussion, we have the following implications on equivalence checking using BCG and PCN. For two models, say M and M', to be functionally equivalent, they must have

1. A one to one mapping of leaf level behaviors,
2. A one to one mapping of observed variables, and
3. Identical firing order for any two behaviors with data dependence.

9.2 Graph Reduction

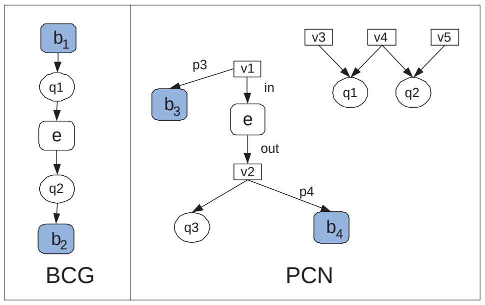
For building an automated tool for equivalence checking, we need to define methods for reducing two models to a *normal form* representation. The reduction procedure must preserve the functionality of the model, as per the above notion. If the normal form of two models is identical, we can claim that the models are functionally equivalent. Otherwise, the result is inconclusive.

Let us now consider some functionality preserving transformations to a model that will lead to its normal form. We will perform these transformations on the BCG and PCN representations of the model. We choose these graph representations to demonstrate the transformations for sake of clarity. The transformations can also be shown on corresponding MA expressions, since the two representations have one-to-one mapping.

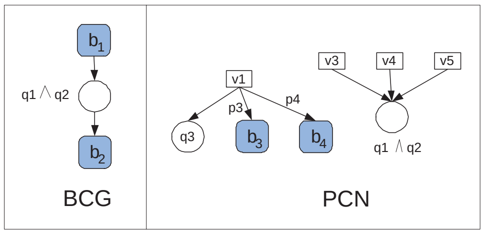
Our goal is to eliminate identity behavior nodes and redundant dependencies from the BCG and PCN, as the model is reduced to its normal form. Redundant dependencies include control dependencies that do not influence the value trace of the observed variables.

9.2.1 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.



(a) Before applying identity elimination



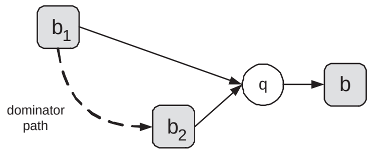
(b) After applying identity elimination

Figure 22. Parts of BCG and PCN before and after identity elimination

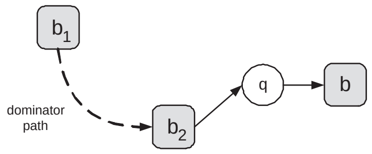
The simple example illustrated in figure 22(b) shows parts of the BCG and PCN involving an identity behavior e. In the BCG, e is part of the control path from b_1 to b_2. It must be noted that there are no other edges to either e or the control nodes q_1 and q_2. As per the semantics of BCG, we can eliminate e by merging the control nodes q_1 and q_2 as shown for the BCG in figure 22(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 22(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 22(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 (R1)

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, $\forall i, j, s.t. 1 \le i \le m, 1 \le j \le n$
In BCG(M), q_i has in-degree $l(i)$ and q'_j has in-degree $k(j)+1$.
Let, $(x^i_1, q_i), (x^i_2, q_i), \ldots, (x^i_{l(i)}, q_i) \in E(BCG(M))$, and $(e, q'_j), (y^j_1, q'_j), \ldots, (y^j_{k(j)}, q'_j) \in E(BCG)$ 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, $\forall i, j, s.t. 1 \le i \le m, 1 \le j \le n$
$q_i \wedge q'_j : x^i_1 \& \ldots \& x^i_{l(i)} \& y^j_1 \& \ldots \& y^j_{k(j)} \rightsquigarrow z_j \in BCG(M')$
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.$(v', x, p) \in PCN(M)$
$PCN(M') = (PCN(M) - (v', x, p)) \cup (v, x, p)$.



(a) BCG before redundant control elimination



(b) BCG after redundant control elimination

Figure 23. BCG before and after redundant control elimination

9.2.2 Redundant control dependency elimination

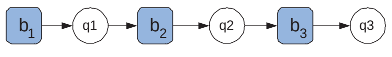
In order to eliminate spurious edges in a BCG, we first need a control dependence analysis. Given model M, let $y \in N_B(BCG(M)), x \in N(BCG(M))$. If during any execution of M, y **always** fires at least once between every firing of x, then we define y to be a **dominator** of x. The set of dominator 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 $dom(x,M) = dom(x,M) \cup \bigcup_{(b,x) \in E(BCG(M))} \{b \cup \{y : y \in dom(b,M)\}\}$

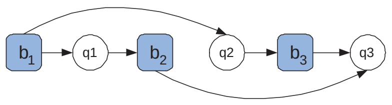
An instance of control dependency elimination is shown in Figure 23. Given $q \in N_Q(BCG(M))$. Let
$b_1, b_2 \in N_B(BCG)$ and $(b_1,q),(b_2,q) \in E(BCG)$
Thus b_1 and b_2 must fire for q to fire.If we can show that $b_1 \in dom(b_2,M)$ then the edge (b_1,q) can be eliminated from the BCG. This is 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.

Redundant 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
$E(BCG(M)) = E(BCG(M)) - (b_1,q)$.



(a) BCG before edge relaxation



(b) BCG after relaxation of edge (b2, q2)

Figure 24. Control relaxation for edge (b_2,q_2)

9.2.3 Control relaxation

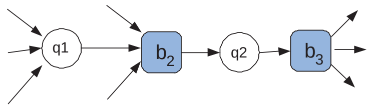
Figure 24 illustrates the control relaxation rule. Given model M, let (b_2,q_2) and (q_2,b_3) be edges in the BCG of M. If there is no data dependency between b_2 and b_3 and between b_2 and q_2, then changing the order of firing between b_2 and q_2, or b_2 and b_3 would not change the value trace for any variable in M. Therefore, the artificial control dependency from b_2 to q_2 may be removed, as illustrated in figure 24. 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 24 is functionally correct under the given restrictions.

Control relaxation rule (R3)

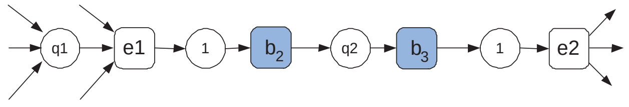
Given model M, let
$(b_2,q_2),(q_2,b_3),(b_1,q_1),(q_1,b_2),(b_3,q_3) \in E(BCG(M))$.
If the following conditions hold

1. $\nexists b \neq b_1$, s.t. $(b,q_1) \in E(BCG(M))$, **and**
2. $\nexists q \neq q_1$, s.t. $(q,b_2) \in E(BCG(M))$ **and**
3. $\nexists b' \neq b_3$, s.t. $(b',q_3) \in E(BCG(M))$ **and**
4. $\nexists v,p,p' \in N(PCN(M))$, s.t.
$(b_2,v,p),(v,q_2),(v,b_3,p') \in E(PCN(M))$

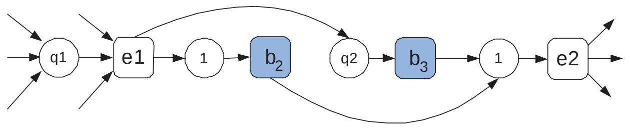
then
$E(BCG(M)) = E(BCG(M)) \cup \{(b_1,q_2),(b_2,q_3)\} - (b_2,q_2)$.



(a) Original BCG without in/out degree restrictions



(b) BCG after addition of identity behaviors e1 and e2



(c) BCG after relaxation of edge (b2, q2)

Figure 25. 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 25. 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 25. Finally, after control relaxation, the identity reduction rule can be applied to optimize away e_1 and e_2.

9.3 Comparison of MA Models

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. In the following sections, we will look at the verification requirements at each stage of system level design. We will also present the application of our equivalence checking method for some of these design steps using simple examples.

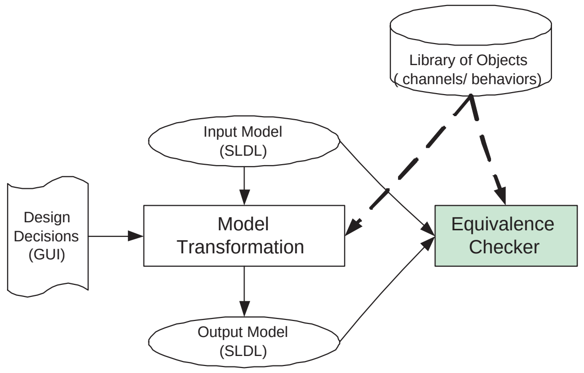


Figure 26. Automatic equivalence checking of system level models

10 System Level Verification Methodology

Figure 26 shows the methodology for generating a refined SLDL model from the input and checking the functional equivalence of the two models. The model generation algorithm uses the design decisions and syntactically transforms the input model. The transformation essentially consists of rearrangement and/or replacement of objects in the input model to create an output model.

Each of the design decisions result in different types of transformations. For different types of transformations, we need a different verification technique to validate it. We will follow the system level design methodology, as shown in Figure 1. The following design steps are encountered as we start from a functional specification model and produce a scheduled transaction level model.

1. Behavior partitioning
2. Static scheduling
3. RTOS insertion
4. Bus insertion

We will now look at the model transformations resulting from these design decisions and the requirements for verifying those transformations.

10.1 Behavior Partitioning

A given specification consists of an arbitrary hierarchy of behaviors. During partitioning, we determine the number of PEs that will be needed to implement the design. The leaf behaviors in the specification are then distributed over these PEs. The PEs are assumed to execute concurrently. Thus, in this step, the design decision is to map each leaf behavior in the specification model to a PE.

The output model must follow a well defined template to reflect the mapping decision. The output shows the PEs as a parallel composition of hierarchical behaviors. Each PE behavior is composed from the leaf level behaviors that were mapped to it. Hence, the transformation produces a rearrangement of behaviors. Additional channels are added from the library for synchronization amongst behaviors. We need this synchronization since the original order of execution of the leaf behaviors must be maintained in the new model as well. The data flow relations in the original model must also be modified to reflect the locality of memory in each PE. The original data transfers between leaf behaviors, mapped to different PEs, will now go across PEs. Hence, they must be routed via identity behaviors using channels.

Figure 27 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. 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.

Since the transformations consist of rearrangements and addition of identity behaviors and channels, equivalence

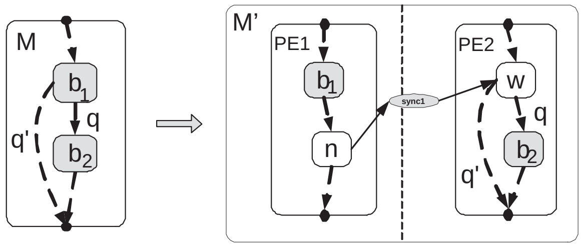


Figure 27. Model generation after behavior partitioning

checking would work for this transformation. Note that in this step, we do not add any new non-identity leaf behaviors to the model. Hence, the equivalence checker can resolve the new objects in the model.

10.2 Static Scheduling

Static scheduling is performed in system level models either due to resource constraints or timing optimization. Behaviors mapped to HW are typically targeted for implementation with a single controller. As a result, any parallelism in the HW PEs must be serialized statically. 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 are also possible as long as they do not violate data dependencies.

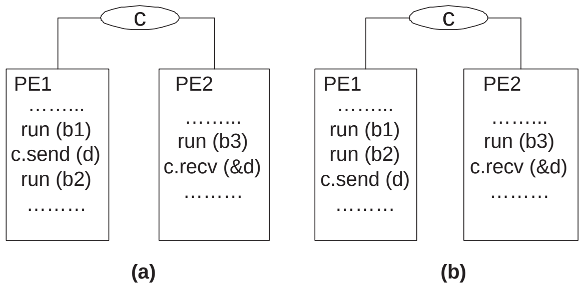


Figure 28. Different communication schedules for transaction over channel *c*.

Reordering of behaviors can also take place as a result of communication scheduling. Such a scenario is shown in Figure 28, 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 28(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 28(b), optimizes timing.

Since, the static scheduling process changes only the control dependencies, the leaf level behaviors in input and output models should match. As in partitioning, we do not add any new non-identity leaf behaviors to the model. Therefore, equivalence checking is also applicable in this design step.

10.3 RTOS Insertion

As explained above, static scheduling is one way of serialization of behaviors in a model. However, PEs that implement software may provide for dynamic scheduling. In this case, the non-determinism of concurrency is resolved at execution time. In a parallel composition of behaviors, the ordering of behavior execution is done during run time. This ordering is performed by a scheduler that is part of the PE's operating system. In a SLDL implementation, the scheduler is another behavior that models the Real Time Operating System (RTOS). The concurrent behaviors are then modified to make calls to the OS behavior, before starting execution. If a behavior is scheduled for execution, it is notified by the scheduler. Upon completion, the behavior must notify its completion to the RTOS and release the PE resources. Competing scheduler requests are resolved using a scheduling policy. Most of these policies are well known like EDF, Round Robin, FIFS etc. Essentially, during RTOS insertion, the dynamic scheduling policy of the SLDL simulator is replaced by that of the explicit RTOS.

The addition of a new non-identity behavior (in this case, the scheduler) renders our equivalence checker unusable. Therefore, for dynamic scheduling, we need to ensure that the scheduler behavior, and hence the scheduling policy, satisfies the same properties as the scheduler of the SLDL simulator. This can be verified using a property verification tool. If the property verification is successful, the scheduler behavior can be abstracted away from the model. The MA expression during this SLDL transformation can, thus, remain unchanged if the dynamic scheduler of the RTOS follows the simulator's properties.

10.4 Bus Insertion

After behavior partitioning and scheduling, the system model consists of concurrent behaviors communicating with several channels. Although, the model shows the computation structure correctly, the communication structure still needs to be implemented. In a bus-based SoC communication scheme, the various PEs are connected to system busses. The communication model can thus be represented using channels for busses. All virtual links in the input model are shared over the new *bus channels*. The design decision in this case is choosing the number of bus channels and mapping the virtual links to the bus channels. Also, the ordering of transactions on the bus may be done using an arbitration policy. By default, in MA, we follow a first-come first-serve policy as explained in Section 7.2. However, if the designer chooses to use a different scheduling, he or she may add a new *arbiter behavior* to schedule transactions over a bus. The arbitration of transactions is analogous to dynamic scheduling of behaviors, described above.

For verification, we have the same scenario as that in RTOS insertion. Equivalence checking is not directly applicable since the output model has a new non-identity leaf behavior. As before, we can use property checking to verify that the arbitration policy preserves the functionality of the model. If we can prove, using a property checking tool, that the arbiter behavior will

1. never cause a deadlock
2. eventually schedule a transaction request

then we can abstract it away.

11 Case Study: Verification of Behavior Partitioning

In this section, we will use a simple example to demonstrate functional equivalence checking of models before and after behavior partitioning. We start by capturing the specification and (partitioned) architecture model as hierarchical behaviors in MA. The models are flattened using the flattening laws described in Section 6.3. Then, we resolve the channels and derive the BCG and PCN for the two models. Finally, we use the reduction rules in Section 9.2 to obtain the normal form graphs for the models. The isomorphism of the normal form graphs is used to check the equivalence of the specification and architecture models.

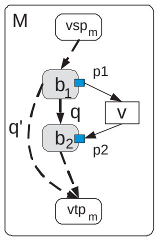
11.1 Specification and Architecture Models

The specification model M, shown in figure 29 is composed of two leaf level behaviors b_1 and b_2 and variable v, which is written by b_1 and read by b_2. The following pseudo code describes the specification

M: **begin**;**run** b_1; **if** $v < 5$ **run** b_2;**end**

Assume that b_1 uses port p_1 to write v and b_2 uses port p_2 to read v. Also, let $q = v < 5$ and $q' = v \geq 5$. The MA expression for M can be written as follows:

$M = [vsp_m].[vtp_m].[b_1].[b_2].b_1 < p_1 > \longrightarrow v.v \rightarrow b_2 < p_2 > .vsp_m \rightsquigarrow b_1.q : b_1 \rightsquigarrow b_2.q' : b_1 \rightsquigarrow vtp_m.b_2 \rightsquigarrow vtp_m$



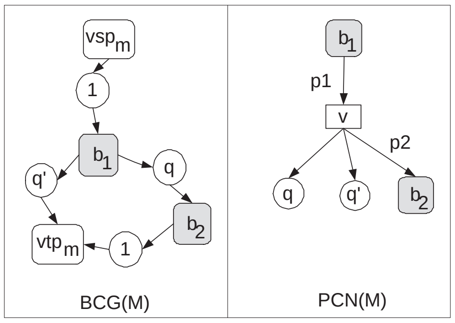


Figure 29. Example of a simple specification model and its graph representations

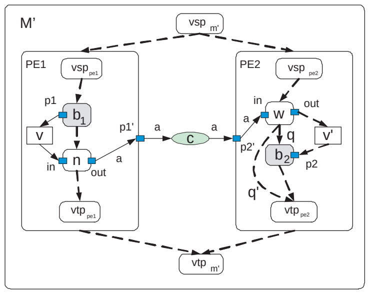


Figure 30. Architecture model derived after partitioning

Let us assign two PEs, namely PE1 and PE2 to implement this specification. Also, let us map b_1 to $PE1$ and b_2 to $PE2$. A possible architecture model M', created from the specification and the mapping decision, is shown in Figure 30. Note that additional hierarchy is created for PE1 and PE2, which execute in parallel. New identity behaviors n and w are introduced in PE1 and PE2, respectively. These identity behaviors are used to send data in variable v from PE1 to PE2, via the channel c. This data is needed by b_2 and the control conditions q and q'. Note that the data from v is copied into variable v', which is local to PE2. In MA, the architecture model can be written using the following expressions.

$M' = [vsp_{m'}].[vtp_{m'}].[pe_1].[pe_2].vsp_{m'} \rightsquigarrow pe_1.$
$vsp_{m'} \rightsquigarrow pe_2.pe_1 \& pe_2 \rightsquigarrow vtp_{m'}.$
$c < a >: pe_1 < p'_1 > \mapsto pe_2 < p'_2 >$

$pe_1 = [vsp_{pe_1}].[vtp_{pe_1}].[b_1].[n].vsp_{pe_1} \rightsquigarrow b_1.b_1 \rightsquigarrow n.$
$n \rightsquigarrow vtp_{pe_1}.b_1 < p_1 > \rightarrow v.v \rightarrow n < in > .$
$< a >: n < out > \mapsto I < p'_1 >$

$pe_2 = [vsp_{pe_2}].[vtp_{pe_2}].[w].[b_2].vsp_{pe_2} \rightsquigarrow w.$
$q : w \rightsquigarrow b_2.q' : w \rightsquigarrow vtp_{pe_2}.b_2 \rightsquigarrow vtp_{pe_2}.$
$< a >: I < p'_2 > \mapsto w.w < out > \rightarrow v'.v' \rightarrow b_2 < p_2 >$

11.2 Flattening of Models

The specification model has only one level of hierarchy, so it cannot be flattened any further. The architecture model, on the other hand, has two levels of hierarchy. Therefore, we can flatten the architecture model to remove the hierarchy created by behaviors pe_1 and pe_2. Using the laws in Section 6.3, the flattened architecture model can be expressed in MA as follows
$M' = [vsp_{m'}].[vtp_{m'}].[vsp_{pe_1}].[vtp_{pe_1}].[b_1].[n].$
$[vsp_{pe_2}].[vtp_{pe_2}].[w].[b_2].$
$vsp_{m'} \rightsquigarrow vsp_{pe_1}.vsp_{m'} \rightsquigarrow vsp_{pe_2}.vtp_{pe_1}\&vtp_{pe_2} \rightsquigarrow vtp_{m'}.$
$vsp_{pe_1} \rightsquigarrow b_1.b_1 \rightsquigarrow n.n \rightsquigarrow vtp_{pe_1}.$
$vsp_{pe_2} \rightsquigarrow w.q : w \rightsquigarrow b_2.q' : w \rightsquigarrow vtp_{pe_2}.$
$b_2 \rightsquigarrow vtp_{pe_2}.b_1 < p_1 > \rightarrow v.v \rightarrow n < in > .$
$w < out > \rightarrow v'.v' \rightarrow b_2 < p_2 > .$
$c < a >: n < out > \mapsto w < in >$

11.3 Reduction to Normal Form

We will now use the reduction rules in Section 9.2 to derive the normal form representation of the two models. Let us start by considering the architecture model. Channel c in the flattened MA expression of M' can be resolved into edges in the BCG and PCN as described in Section 7.3.

Figure 31 shows how the BCG and PCN for the architecture model shown M' are reduced to its normal form in a step wise fashion. The BCG is the graph shown on the left and the PCN is the graph shown on the right. At each step, we use the applicable reduction rule until we cannot apply them any more. The topmost BCG and PCN in Figure 31 are derived from the architecture model. The resulting control dependencies from resolution of channel c are seen in the edges emanating from n and w in the BCG. Also, the edge (n,w) in the PCN indicated the channel connection.

The reduction to normal form takes place as follows. In the first step, we optimize away identity behaviors vsp_{pe1} and vsp_{pe2} from the BCG using identity elimination rule R1. There is no change made to the PCN since these behaviors do not have any data dependencies. In Step 2, we use R1 again to optimize away node n. As a result, in the BCG, all edges emanating from n are replaced by those from b_1. In the PCN, this results in the node n and its edges being removed. A new edge (v,w,in) is added to the PCN to indicate that w now reads directly from v via its in port. Similarly, in step 3, node w is optimized away using R1. Hence, all edges from w are replaced by those from $vsp_{m'}$ in the BCG. In the PCN, this reduction results in node w and v' being optimized away. All edges from v' are now converted to edges from v.

Step 4 uses redundant control elimination rule R2 to get rid of redundant edges from $vsp_{m'}$ in the BCG. It may be noted that by definition of dominator in Section 9.2.2, we have $vsp_{m'} \in dom(b_1, M')$ Note that in the BCG after Step 3, 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 $(vsp_{m'},q')$. There are no changes to the PCN.

In Step 5, we use R1 once again to eliminate identity behaviors $vtppe1$ and vtp_{pe2} from the BCG. Again, the PCN remains unchanged since these behaviors do not have any data dependencies. 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 $b_1 \in dom(b_2, M')$. Again, the PCN remains unchanged. The BCG and PCN obtained after Step 6 cannot be reduced any further and, thus, represent the normal form for the architecture model M'.

The normal form graphs of M' are identical to the BCG and PCN of corresponding specification model M, shown in Figure 29(b). Hence, we have shown the function equivalence of the two models before and after mapping of specification behaviors to PEs for our example.

12 Related Work

Significant research has been done in the past for developing modeling formalisms for system level design. Process algebras, such as CSP [8] and CCS [12] have been used for verifying distributed software, but have limitations in modeling. For example, CSP allows only rendezvous communication between parallel processes. StateCharts [7] provide for hierarchy, synchronization and exceptions, but have unclear execution semantics, which have led to several variants. Colored Petri Nets are widely used for analysis and modeling of concurrent systems, and verification techniques have been developed to check for their equivalence [9]. Formal methods, developed for hardware verification, have been applied to embedded systems like bounded model checking [4] and theorem proving [13]. The problem with most state based approaches, as above, is that their complexity increases exponentially with design size. Our goal is to correctly derive detailed system level models, so that we can leave the functional verification task for only the specification model. Correct by construction techniques have been widely applied at RT Level to prove the correctness of high level synthesis steps [13] [3]. A complete methodology for correct digital design has been proposed in [11], but they

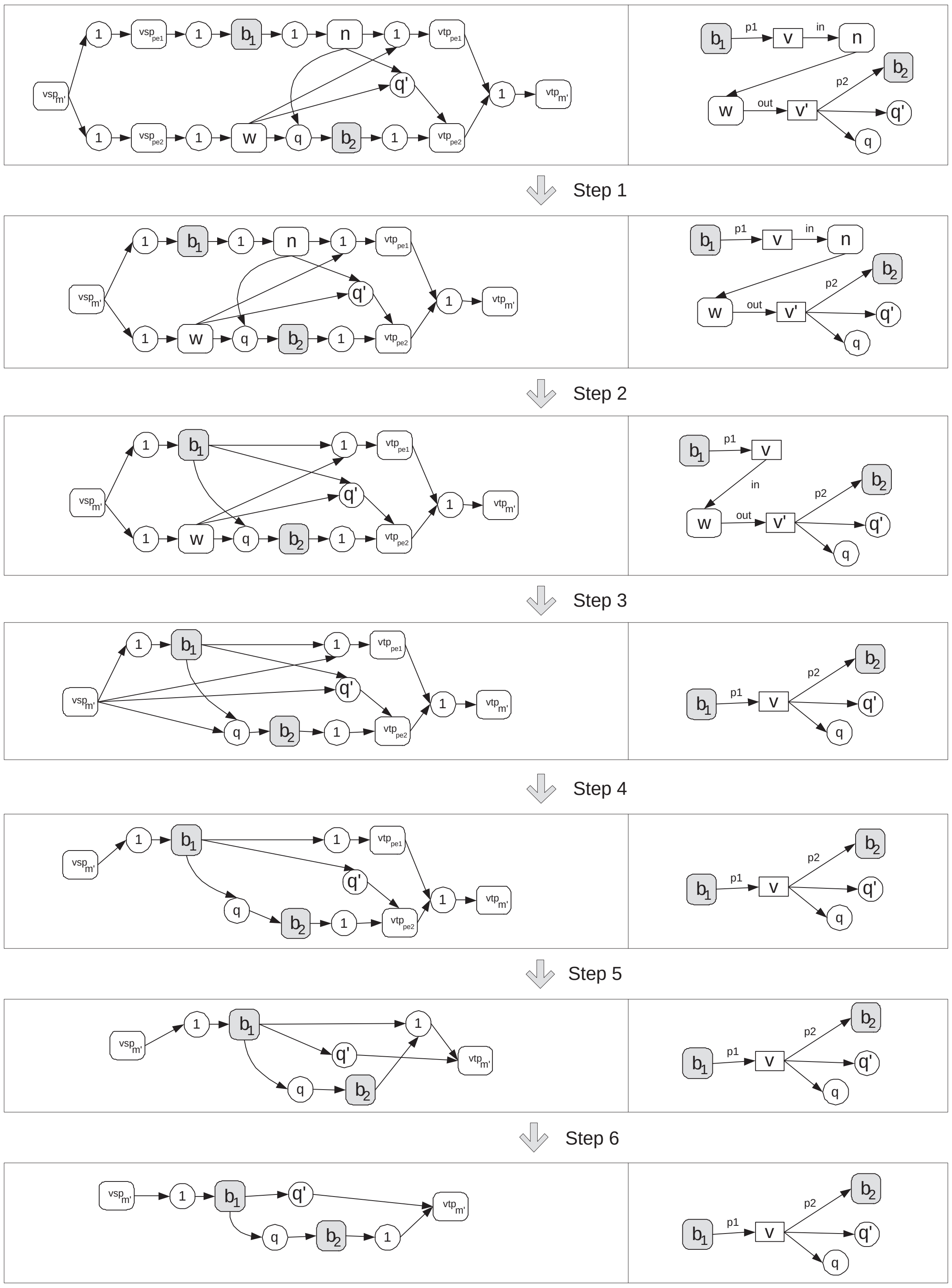


Figure 31. Reduction of architecture model BCG, PCN pair to normal form

only consider synchronous models which are insufficient at system level.

More recently, research is being directed towards comparison of SLDL models using textual correlation and symbolic simulation [14], but their approach requires two models to be very similar. Verification of only the synchronization primitives of SpecC [6] are presented in [15]. Correct by construction approaches at the system level have been proposed for HW/SW partitioning [2]and model generation [1], but they restrict the designer to follow a given refinement algorithm.

13 Conclusions

In this paper, we introduced a formalism called Model Algebra, which can be used for functional verification of system level models. The objects and composition rules of Model Algebra allowed us to represent hierarchical SLDL models as expressions. We then presented the execution semantics of model algebraic descriptions using BCG and PCN graphs. We also established a notion of functional equivalence of two models based on the value trace of variables in the models. This led us to define functionality preserving transformation rules on model algebraic descriptions. The expressive power and well defined rules in MA can be used to derive new equivalent models from the specification. On the other hand, these rules can also be used to verify functional correctness of model refinements resulting from system synthesis. We showed how models in MA can be reduced to a normal form, which allowed us to compare the input and output of system level design steps. The formalization of models using Model Algebra has significant impact on system level verification.

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] R. Camposano. Behavior-preserving transformations for high-level synthesis. In *Proceedings of the Mathematical Sciences Institute workshop on Hardware specification, verification and synthesis: mathematical aspects*, pages 106–128. Springer-Verlag New York, Inc., 1990.

[4] X. Chen, H. Hsieh, F. Balarin, and Y. Watanabe. Case studies of model checking for embedded system designs. In *Third International Conference on Application of Concurrency to System Design*, pages 20–28, June 2003.

[5] D. Gajski, R. Domer, A. Gerstlauer, and J. Peng. *System Design with SpecC*. Kluwer Academic Publishers, January 2002.

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

[7] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[8] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[9] J. Jorgensen and L. Kristensen. Verification of colored petri nets using state spaces with equivalence classes. In *Proceedings of the Workshop on Petri Nets in System Engineering*, pages 20–31, September 1997.

[10] G. Kahn. The semantics of a simple language for parallel programming. In *Info. Proc.*, pages 471–475, August 1974.

[11] Middlehoek. A methodology for the design of guaranteed correct and efficient digital systems. In *IEEE International High Level Design Validation and Test Workshop*, November 1996.

[12] R. Milner. *A Calculus of Communicating Systems*. Springer, 1980.

[13] S. Rajan. Correctness of transformations in high level synthesis. In *International Conference on Computer Hardware Description Languages and their Applications*, pages 597–603, June 1995.

[14] 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.

[15] 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.