International Journal of Parallel Programming, Vol. xx, No. x, August 2005 (©2005)

Verification of System Level Model Transformations

Samar Abdi and Daniel Gajski

[1] Center for Embedded Computer Systems, University of California, Irvine, CA 92697. Email: {sabdi,gajski}@cecs.uci.edu

This paper presents Model Algebra (MA), a formalism for representing SoC designs at system level. We define the objects and composition rules of MA and show how system level models can be represented as expressions in this formalism. The formalism is applied to a system level design methodology, where design decisions are used to gradually transform the functional specification model of the system to a transaction level model with components and communication structure. Each transformation is represented as a manipulation of a model algebraic expression, and proven for correctness using the laws of model algebra. These laws are based on the well defined execution semantics and notion of functional equivalence for MA models. Our approach promises significant savings in the verification of system level models because only the first model needs to be verified using conventional techniques. All transformations of this model, derived using MA laws, are proven to be functionally equivalent.

KEY WORDS: System level modeling, Verification, Model transformations, Design methodology.

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 System Level Design Languages (SLDLs), can be abstracted into MA expressions. Model transformations are realized by manipulation of MA expressions. The formalism provides a set of laws that can be used to transform one model into an equivalent model at a different level of abstraction.

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 communication synthesis, the final bus architecture, including the busses and their connections to components, is determined. Also, the abstract point to point traffic between these components is routed on this bus architecture. Finally, the SW tasks are compiled for the target processor and the HW behaviors are synthesized.

The rest of the paper is organized as follows. Section 2 discusses the requirements for modeling at the system level and presents 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 3. In Section 4, we deal with semantics of hierarchy and the impact of granularity on model analysis. The formal execution semantics, includ-

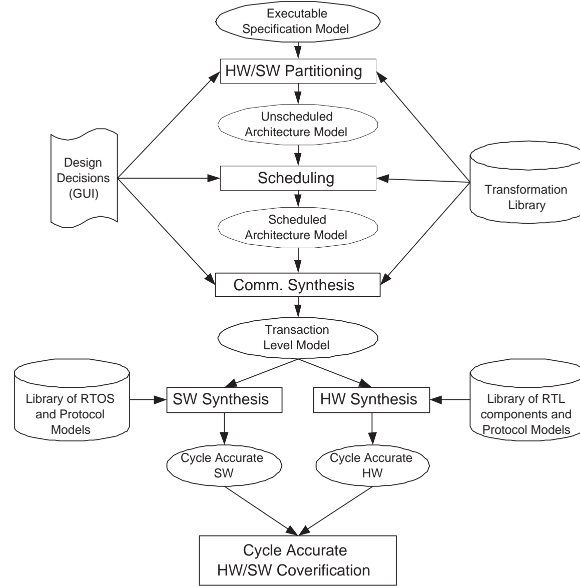


Fig. 1. A possible system level design methodology

ing channel semantics, of models expressed in MA is defined in Section 5 and the functional equivalence verification of MA models is discussed in Section 6. In Section 7, we look at each design step in our system level methodology and discuss verification methods for them using MA. Finally, we give a brief overview of related work in Section 8 and wind up with conclusions.

2. MODEL ALGEBRA

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.

The objects of MA can be defined as the tuple
$< \mathscr{B}, \mathscr{C}, \mathscr{V}, \mathscr{I}, \mathscr{P}, \mathscr{A} >$, where
$\mathscr{B}$ is the set of behaviors
$\mathscr{C}$ is the set of channels
$\mathscr{V}$ is the set of variables
$\mathscr{I}$ is the behavior interface
$\mathscr{P}$ is the set of behavior ports
$\mathscr{A}$ is the set of address labels

We also define a subset $\mathscr{B}^I$ of $\mathscr{B}$ 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 $\mathscr{V}$ has *type* associated with it. We define $\mathscr{Q}$ to be the subset of $\mathscr{V}$ such that all variables in $\mathscr{Q}$ are of type **boolean**.

2.1. Ports

Each behavior has an associated object called its interface. The interface carries the ports of the behavior that are represented by their association to the behavior. Hence, to internal behaviors of a hierarchical behavior, the port is seen as $\mathscr{I} < p >$, where $p \in \mathscr{P}$. The port is treated like any other local variable except that we restrict operations on it, depending on its *direction*. 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*. If both read and write are allowed, th port is called *inout-port*. When the same port p is accessed from outside of behavior b, it is written as $b < p >$.

2.2. 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 written 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.

2.3. 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, in a particular format, to create hierarchical behaviors.

2.3.1. 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 $\forall i, 1 \leq i \leq nb, b_i \in \mathscr{B} \cup \mathscr{I}, q \in \mathscr{Q}$. 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 to complete. 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. A relation with a TRUE condition, eg. $1 : b_1 \rightsquigarrow b_2$ will be shorthanded as $b_1 \rightsquigarrow b_2$.

2.3.2. Non-blocking write

This composition rule (R_{nw}) is used to indicate that a behavior uses its out-port to write 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

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

In this case, the composition rule indicates a port-map in a hierarchical behavior. Note that $< p' >$ must also be an out-port or inout-port.

2.3.3. Non-blocking read

This composition rule (R_{nr}) is used to indicate that a behavior uses its in-port to read 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

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

In this case, the composition rule indicates a port-map in a hierarchical behavior. Note that $< p' >$ must also be an in-port or inout-port.

2.3.4. 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

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

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 a.

2.3.5. 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

$$< a >: b < p > \mapsto \mathscr{I} < p' >$$

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

2.3.6. 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

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

where $b_1 < p_1 >$ through $b_n < p_n >$ are the in-port(s) of the receiving behavior(s). The port $\mathscr{I} < p' >$ 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.

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

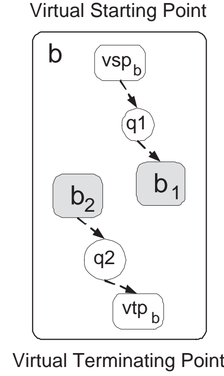


Fig. 2. Control flow within hierarchical behaviors

3. 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 using these objects and composition rules.

3.1. 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. In figure 2, a behavior b (shown as rounded rectangle) is created by hierarchical composition of sub-behaviors b_1 and b_2. 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 (shown using broken arcs and circular nodes labeled with control conditions) from vsp_b. We can see in figure 2 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 to other sub-behaviors of b. Likewise, the identity behavior vtp_b is the last behavior to execute inside b, and will only have incoming control 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

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

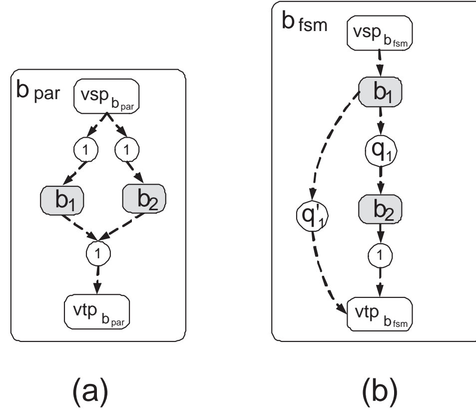


Fig. 3. (a)Parallel and (b)FSM style compositions of behaviors

3.2. Parallel and Conditional Execution

Most SLDLs provide for special constructs to create different types of behavioral hierarchies. The common ones are parallel composition and conditional composition. Figure 3(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 *if-then-else* style composition of behaviors is shown in Figure 3(b). A simple pseudo code example for a hierarchical behavior b_{fsm} is **run** b_1; **if** $q_1 == 1$ **goto** l2 **else** break; The control relations of b_{fsm} can be written as follows in MA
$vsp_{b_{par}} \rightsquigarrow b_1.q_1 : b_1 \rightsquigarrow b_2.q_1' : b_1 \rightsquigarrow vtp_{b_{par}}.b_2 \rightsquigarrow vtp_{b_{par}}$

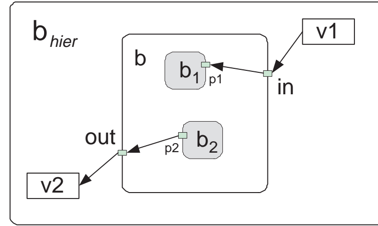


Fig. 4. Using ports for non-blocking data flow in hierarchical behaviors

3.3. Variable Access via Ports

In MA, a variable (shown using rectangular box) 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. Ports are visualized as tiny boxes on the behavior interfaces. As shown in Figure 4, 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 (shown using solid directed arcs) from the interface of b to b_1. We can write this as the relation $\mathscr{I} < in > \rightarrow b_1 < p_1 >$.

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 4. 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 $b_2 < p_2 > \rightarrow \mathscr{I} < out >$.

3.4. Channel Access via Ports

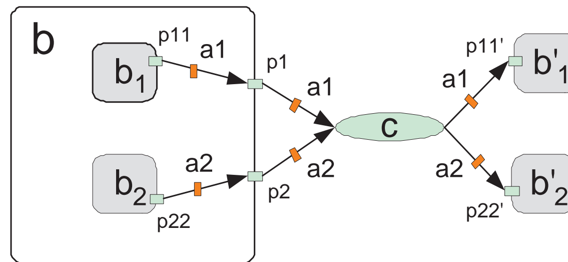


Fig. 5. Sharing channel for transactions with different addresses

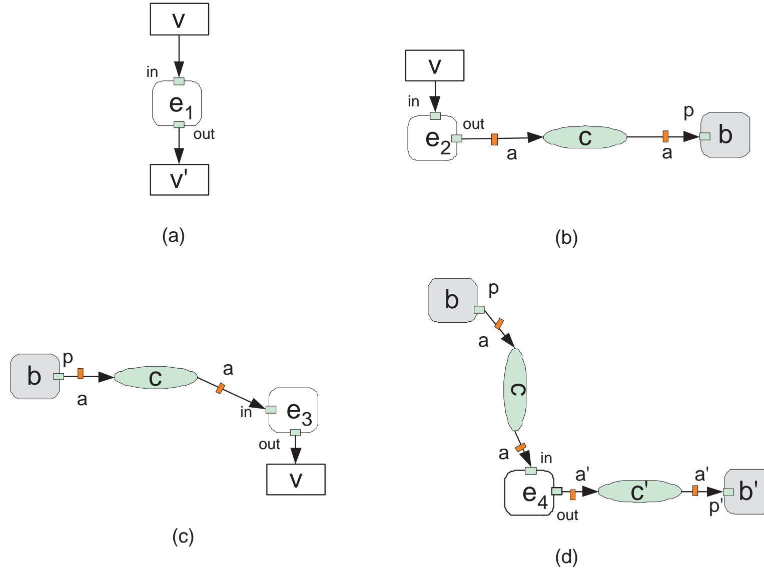


Fig. 6. Various manifestations of the identity behavior

As in the case of non-blocking reads and write, MA provides mechanism for blocking reads and writes (shown using solid arcs labeled with link address) via ports. For instance, in Figure 5, we see channel transactions from b to b'_1 and b'_2 over c, labeled a_1 and a_2 respectively. After zooming into the hierarchy of b, we see that the transactions are taking place from b_1 to b'_1 and b_2 to b'_2. The ports p_1 and p_2 of b makes the channel c visible to b_1 and b_2. Therefore, using the relation $< a >: b_1 < p_{11} > \mapsto \mathscr{I} < p_1 >$. behavior b_1 can access channel c. However, this requires p_1 to be bound to the transaction link addressed by a_1.

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 5 shows an instance of channel sharing. Here, the two virtual links with addresses $< a_1 >$ and $< a_2 >$ use a common channel c. Transactions may be attempted concurrently on these links. However, due to sharing of the channel, we can allow only one transaction at a time. Thus, an arbiter in the channel must ensure that only one transaction may take place at any time. In MA, this is guaranteed by the mutual exclusion property of the channel, 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.

3.5. 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 6. In the first case, as shown in figure 6(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 second case of identity behavior is shown in figure 6(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 third case of identity behavior is shown in figure 6(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 $c < a >: b < p > \mapsto e_3 < in > .e_3 < out > \rightarrow v$. Finally, the fourth manifestation of identity behavior is shown in figure 6(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 $c < a >: b < p > \mapsto e_4 < in > .c' < a' >: e_4 < out > \mapsto b' < p' >$.

4. HIERARCHICAL MODELING IN MA

The model of a system is simply 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.

4.1. Internal and Interface Terms

In MA, it is possible to represent a hierarchical behavior as a grouping of terms involving its sub-behaviors, its interface and its local variables

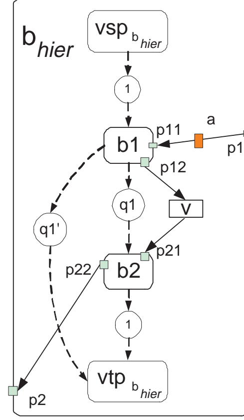


Fig. 7. A hierarchical behavior with local objects and relations

and channels. Figure 7 shows a hierarchical behavior b_{hier}. The expression for the hierarchical behavior is written using the local objects and their compositions. 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 7, we can see that
$|b_{hier}| =< a >: \mathscr{I} < p_1 > \mapsto b_1 < p_{11} > .b_2 < p_{22} > \rightarrow \mathscr{I} < p_2 >$

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

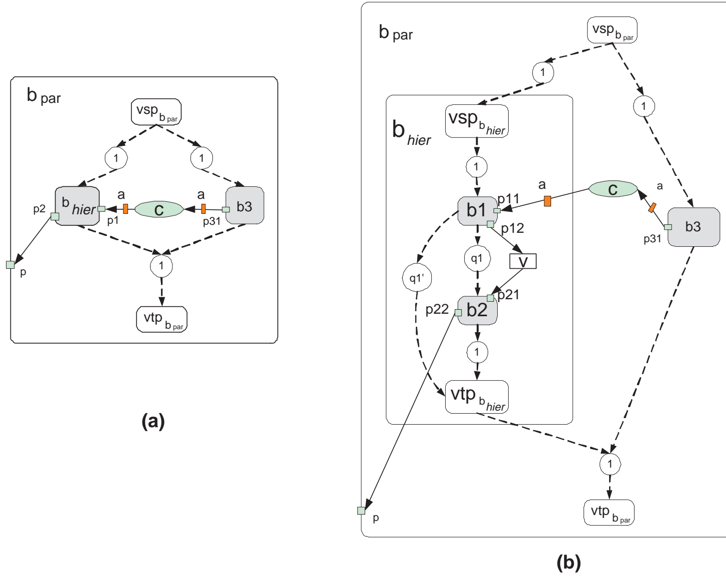


Fig. 8. Hierarchical behavior b_{par} with a parallel composition (a) before, and (b) after flattening

4.2. Multiple Levels of Hierarchy

In the above example, a conditional hierarchical composition was created. The resulting behavior b_{hier} can be used further to create more hierarchical behaviors. For instance, in figure 8(a), 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

$b_{par} = ([vsp_{b_{par}}].[b_{hier}].b_3.[vtpb_{par}].vsp_{b_{par}} \rightsquigarrow b_{hier}.$
$vsp_{b_{par}} \rightsquigarrow b_3.b_{hier}\&b_3 \rightsquigarrow vtpb_{par}.$
$c < a >: b_3 < p_{31} > \mapsto b_{hier} < p_1 > .b_{hier} < p_2 > \rightarrow \mathscr{I} < p >)$

4.3. Flattening of Hierarchical Behaviors

Addition of hierarchy allows the designer to group different behaviors together. It does not add any functionality to the model. 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 7.According to 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, y and z 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 8(b). 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} is removed 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 $(...y \rightarrow \mathscr{I} < p > ...) < p > \rightarrow x \Longrightarrow y \rightarrow x$

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

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

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

5. EXECUTION SEMANTICS

In order to define the execution semantics of MA, the control relations in the model are captured using the *Behavior control graph*(BCG). BCG is simply a graph representation of the control flow in the model. Control dependencies implied by the rendezvous transactions can also converted into BCG arcs.

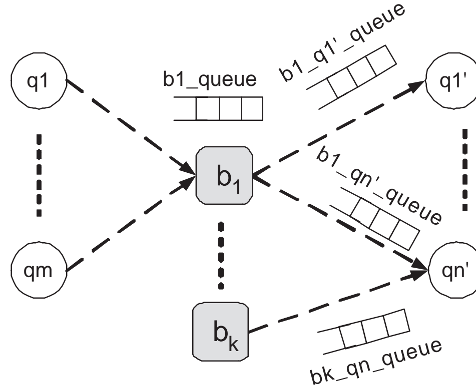


Fig. 9. The firing semantics of BCG nodes

5.1. Behavior Control Graph

The BCG is similar in principle to the Kahn Process Network [(1)], but with some remarkable differences.It is a directed graph BCG(N,E) with two types of nodes, namely *behavior nodes*(N_B) and *control nodes*(N_Q). The behavior nodes, as the name suggests, indicate behavior execution, while the control nodes evaluate control conditions that lead to further behavior executions. Directed edges are allowed from behavior nodes to control nodes and vice versa. Also, a control node can have one, and only one, out going edge. Thus,
$E(BCG) \subset N_B(BCG) \times N_Q(BCG) \cup N_Q(BCG) \times N_B(BCG)$

The execution of a behavior node, and similarly, evaluation in a control node, will be referred to as a *firing*. Node firings are facilitated by tokens that circulate in the queues of the BCG as shown in Figure 9. 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.

There is a one to one correlation between the BCG and the control relations of the MA representation. A relation of the form

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

translates to a BCG with a control node (q), $n+1$ behavior nodes ($b, b_1, ..., b_n$) and $n+1$ directed arcs $((b_1, q), ..., (b_n, q), (q, b))$.

5.2. Channel Semantics

The channel object allows for reliable communication between two concurrently executing behaviors. As discussed before, a channel transaction implies a control dependency between parts of the communicating behaviors. We will assume both the sender and the receiver to be identity behaviors in future discussions.

5.2.1. Channel with single transaction link

Figure 10 shows a transaction taking place over channel c. We can express this transaction MA using the term $c < a >: e_{wr} < out > \mapsto e_{rd} < in >$. 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 channel. Subsequently, e_{rd} reads the data from the channel.

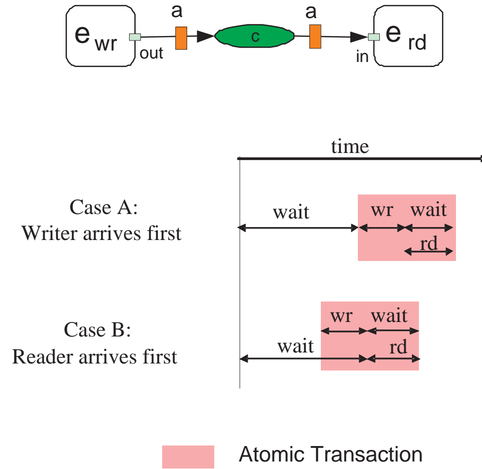


Fig. 10. Timing diagram of a transaction on a channel

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, not during synchronization.

5.2.2. Channels with multiple transaction links

As discussed earlier, channel sharing is possible for different virtual links, but the transactions are ordered in time. This mutual exclusivity of transactions can be achieved by the use of semaphore constructs in a SLDL. Thus, the shaded part representing the actual data read and write over the local memory of channel is mutually exclusive. Consider the configuration shown in figure 11. 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 despite 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 addressed a_2 is ready to be performed before that for a_1. Thus, the

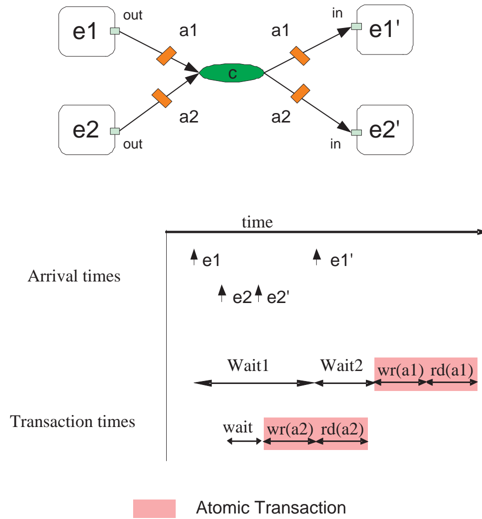


Fig. 11. Multiple competing transactions on a single channel

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 addressed a_2 is in progress.

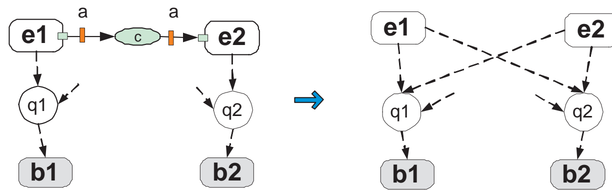


Fig. 12. Resolution of channels into control dependencies

5.2.3. 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 to collect all control dependencies resulting from transaction links and incorporate them into the BCG. We will now see how to resolve the transaction links in flattened MA models into control dependencies. Figure 12 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 12. 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 arc (e_2, q_1) to the BCG. 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 adding the arc (e_1, q_2).

6. EQUIVALENCE OF MODELS

We can ensure the correctness of generated system level models by using transformations proved in MA. Thus, we need a notion of functional equivalence of models in MA. Using this notion, we can define useful laws of MA and prove their soundness.

6.1. 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

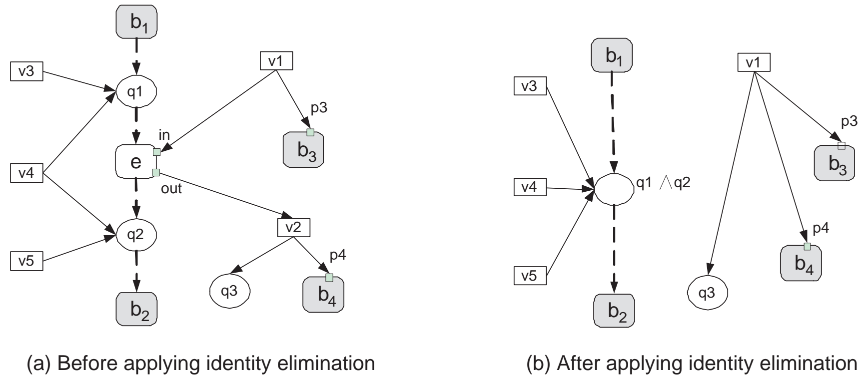
$\forall v \in N_V(PCN(M)), \exists wr(v) \in N_B(PCN(M))$
Let $d_i, i > 0$ be the value written to v after the ith execution of $wr(v)$. Let d_0 be the initial assignment value of v. We define the ordered set
$\tau(v, M, I(M)) = \{d_0, d_1, d_2, ...\}$
We claim that two models M and M' are equivalent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

6.2. Transformation laws of MA

We will now define laws of MA that will allow us to perform useful functionality preserving transformations on a model. For clarity, We will demonstrate these transformations on the graphical representations of the model. The transformations can also be shown on corresponding MA expressions, since the two representations have one-to-one correlation. Due to page restrictions, we will refer the reader to our technical report [(2)] for detailed soundness proofs of these transformations.

6.2.1. Identity elimination

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



(a) Before applying identity elimination (b) After applying identity elimination

Fig. 13. Parts of the model before and after identity elimination

The simple example illustrated in figure 13(b) shows parts of a model involving an identity behavior e, which 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 the model, we can eliminate e by merging the control nodes q_1 and q_2 as shown in figure 13(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 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 model shown in figure 13(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 law (IE)

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 M. Also, let there be n edges from e to control nodes labeled q'_1 through q'_n in M. Now, $\forall i, j, s.t. 1 \leq i \leq m, 1 \leq j \leq n$
In M, q_i has in-degree $l(i)$ and q'_j has in-degree $k(j)+1$.
Let, $(x^i_1, q_i), (x^i_2, q_i), ..., (x^i_{l(i)}, q_i) \in E(BCG(M))$, and
$(e, q'_j), (y^j_1, q'_j), ..., (y^j_{k(j)}, q'_j) \in E(BCG)$ Also, let $(q'_j, z_j) \in E(BCG)$. After, elimination of e, the merger of control nodes would result in $m \times n$ new control nodes. Therefore,
$\forall i, j, s.t. 1 \leq i \leq m, 1 \leq j \leq n, q_i \wedge q'_j : x^i_1 \& ... \& x^i_{l(i)} \& y^j_1 \& ... \& y^j_{k(j)} \rightsquigarrow z_j \in BCG(M')$
For data dependencies, if $(c < a >: e' < out > \mapsto e < in >).e < out > \rightarrow v) \in M, e, e' \in \mathscr{B}^I$, then $M = (M - (c < a >: e' < out > \mapsto e < in >).e < out > \rightarrow v)).e' < out > \rightarrow v$
If $(v \rightarrow e < in > .e < out > \rightarrow v') \in M$, then $\forall x$, s.t.$v' tox < p > \in M, M = (M - v' tox < p >).v \rightarrow x < p >$.

6.2.2. Redundant control dependency elimination

In order to eliminate spurious control dependencies, we first need to do control dependence analysis. Given model a 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

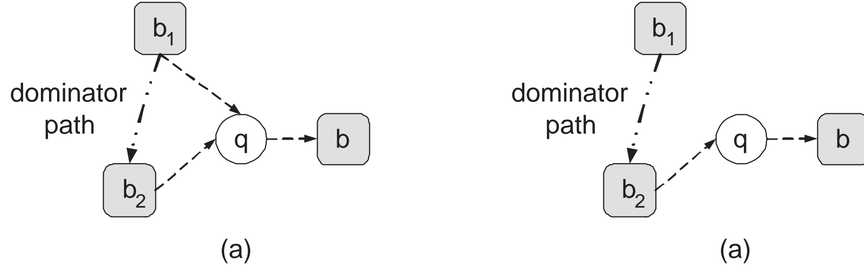


Fig. 14. Model before(a) and after(b) redundant control elimination

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 14. 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 law (RCDE)

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

6.2.3. Control relaxation

Given a model M, let $q : b_1 \rightsquigarrow b_2$ be a control relation in M. If there is no data dependency between b_1 and b_2 and between b_1 and q, then changing the order of execution between b_1 and b_2 would not change the value trace for any variable in M. Therefore, the redundant control relation $q : b_1 \rightsquigarrow b_2$

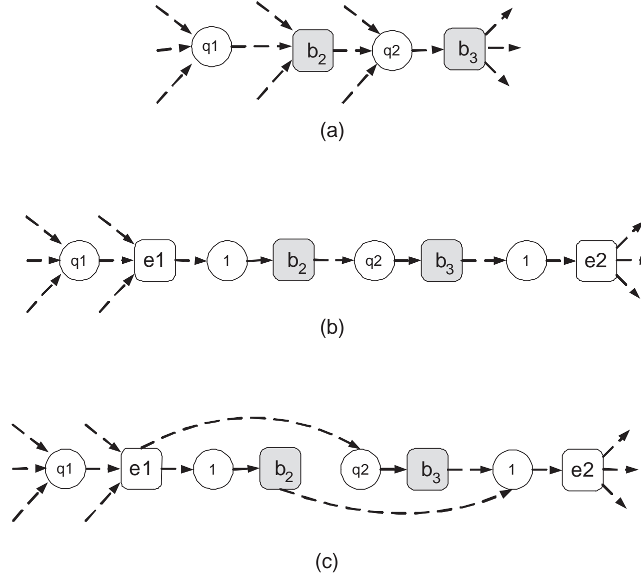


Fig. 15. Control relaxation for $q : b_1 \rightsquigarrow b_2$ after adding e_1 and e_2

may be relaxed as follows. The dependency from b_1 to b_2 is modified into control dependencies from all predecessors of b_1 to b_2, **and** from b_1 to all successors of b_2. In order to simplify the description of this rule, we will massage the original model, shown in Figure 15(a), by adding one identity behavior e_1 before b_1 and one identity behavior e_2 after b_2. The addition of these identity behaviors conforms to the inverse of the identity elimination rule and is, hence, functionality preserving. The massaged model is shown in Figure 15(b). The control relaxation is performed by removing the arc (b_1, q) and adding arcs (e_1, q) and $(b_1, 1)$ as shown in figure 15(c).

Control relaxation law (CR)

Given model M, let $q : b_1 \rightsquigarrow b_2 \in M$. Assuming the configuration shown in 15(b),
If $\nexists v, p_1, p_2$, s.t. $(b_1 < p_1 > \rightarrow v.v \rightarrow b_2 < p_2 > \in M$ **or** q depends on v,
then $M = (M - (q : b_1 \rightsquigarrow b_2)).q : e_1 \rightsquigarrow b_2.b_2 \rightsquigarrow e_2$

7. SYSTEM LEVEL VERIFICATION METHODOLOGY

In our system level design methodology, 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. Communication synthesis

We will briefly discuss the model refinements resulting from these design decisions and the transformation rules required for expressing those refinements. The detailed examples, demonstrating the refinements as a sequence of model transformations, are available in our technical report [(2)].

7.1. 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 16 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'

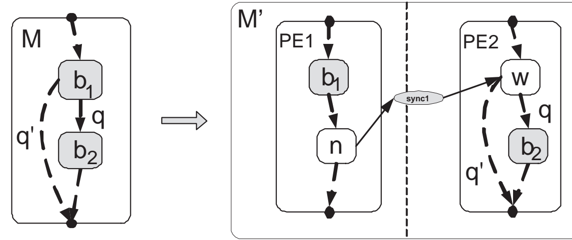


Fig. 16. Model generation after behavior partitioning

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, they can be proved using identity elimination, flattening and redundant control elimination laws of MA.

7.2. 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.

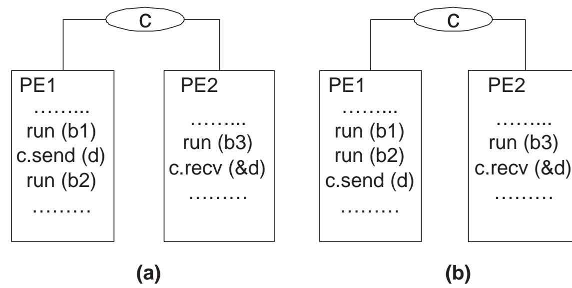


Fig. 17. Different communication schedules for transaction over channel *c*.

Reordering of behaviors can also take place as a result of communication scheduling. Such a scenario is shown in Figure 17, 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 17(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 17(b), optimizes timing. Transformations resulting from static scheduling can be proved using the identity elimination and control relaxation laws of MA.

7.3. Communication Synthesis

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 transaction 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 transaction links to the bus channels. In some cases, a transaction link may need to be implemented on several busses. This will require the addition of new identity behaviors to act as bridges, that will allow the routing of transactions over busses. Hence, the laws for identity elimination and control relaxation can be applied to prove correctness of communication synthesis.

8. RELATED WORK

Significant research has been done in the past for developing modeling formalisms for system level design. Process algebras, such as CSP [3] and CCS [4] have been used for verifying distributed software, but have limitations in modeling. For example, CSP allows only rendezvous communication between parallel processes. StateCharts [5] 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 [6]. Formal methods, developed for hardware verification, have been applied to embedded systems

like bounded model checking [7] and theorem proving [8]. 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 [8] [9]. A complete methodology for correct digital design has been proposed in [10], but they 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 [11], but their approach requires two models to be very similar. Verification of only the synchronization primitives of SpecC [12] are presented in [13]. Correct by construction approaches at the system level have been proposed for HW/SW partitioning [14]and model generation [15], but they restrict the designer to follow a given refinement algorithm.

9. 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 formal execution semantics of model algebraic descriptions using behavior control 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 and perform correct transformations on them. The formalization of models using Model Algebra has significant impact on system level verification.

REFERENCES

1. G. Kahn, The semantics of a simple language for parallel programming., *Info. Proc.*, pp. 471–475 (August 1974).
2. S. Abdi and D. Gajski, *System Level Verification with Model Algebra*, Technical Report CECS-TR-04-29, University of California, Irvine (2004).
3. C. Hoare, *Communicating Sequential Processes*, Prentice Hall (1985).
4. R. Milner, *A Calculus of Communicating Systems*, Springer (1980).

5. D. Harel, Statecharts: A Visual Formalism for Complex Systems, *Science of Computer Programming*, **8**(3):231–274 (June 1987), URL `citeseer.nj.nec.com/harel87statecharts.html`.
6. J. Jorgensen and L. Kristensen, Verification of Colored Petri Nets Using State Spaces with Equivalence Classes, *Proceedings of the Workshop on Petri Nets in System Engineering*, pp. 20–31 (September 1997).
7. X. Chen, H. Hsieh, F. Balarin, and Y. Watanabe, Case Studies of Model Checking for Embedded System Designs, *Third International Conference on Application of Concurrency to System Design*, pp. 20–28 (June 2003).
8. S. Rajan, Correctness of Transformations in High Level Synthesis, *International Conference on Computer Hardware Description Languages and their Applications*, pp. 597–603 (June 1995).
9. R. Camposano, Behavior-preserving transformations for high-level synthesis, *Proceedings of the Mathematical Sciences Institute workshop on Hardware specification, verification and synthesis: mathematical aspects*, pp. 106–128, Springer-Verlag New York, Inc. (1990).
10. Middlehoek, A Methodology for the design of Guaranteed Correct and Efficient Digital Systems, *IEEE International High Level Design Validation and Test Workshop* (November 1996).
11. H. Saito, T. Ogawa, T. Sakunkonchak, M. Fujita, and T. Nanya, An Equivalence Checking Methodology for Hardware Oriented C-based Specifications, *IEEE International High Level Design Validation and Test Workshop*, pp. 274–277 (October 2002).
12. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers (January 2000).
13. T. Sakunkonchak and M. Fujita, Verification of Synchronization in SpecC Description with the Use of Difference Decision Diagrams, *Proceedings of the Forum for Design Languages* (September 2002).
14. E. Barros and A. Sampaio, Towards Provably Correct Hardware/Software Partitioning Using Occam, *Proceedings of the International Workshop on Hardware-Software Codesign*, pp. 210–217 (June 2004).
15. S. Abdi and D. Gajski, Automatic Generation of Equivalent Architecture Model from Functional Specification, *Proceedings of the Design Automation Conference* (June 2004).