# A Formalism for Functionality Preserving System Level Transformations

Samar Abdi

Center for Embedded Computer Systems
UC Irvine
Irvine, CA 92697
Tel: 949-824-8059
Fax: 949-824-8919
e-mail: sabdi@cecs.uci.edu

Daniel Gajski

Center for Embedded Computer Systems
UC Irvine
Irvine, CA 92697
Tel: 949-824-4155
Fax: 949-824-4155
e-mail gajski@uci.edu

**Abstract— With the rise in complexity of modern systems, designers are spending a significant time on modeling at the system level of abstraction. This paper introduces Model Algebra, a formalism built on top of system level design languages, that can be used for implementing functionality preserving transformations on system level models. Such transformations enable us to implement high level design decisions without having to write new models for each design decision. Moreover, since these transformations preserve functionality, the transformed models do not need to be re-verified. We present the definition of Model Algebra and show how system level models can be represented as expressions in this formalism. The laws of Model Algebra are use to define correct model transformations. We show a system level design scenario, where design decisions gradually refine the functional model of the system to an architectural model with components and communication structure. The refinement can be performed using the correct model transformations in our formalism.**

## I. Introduction

Due to the complexity of modern systems, designers have to describe them at a high level of abstraction using system level design languages (SLDLs).However, independently written models in SLDLs have little correlation between them which makes it intractable to compare them for equivalence. A possible solution is to derive the detailed system level models from the specification by a series of functionality preserving transformations. The specification is captured using an executable model, which is a purely functional description of the design. As design decisions are made during system synthesis, the designer refines the functional specification model to a more detailed model representing the target architecture. Such a design methodology requires formalisms to represent the system level models and notions for their correctness refinement. In this paper, we introduce **Model Algebra** (MA), which is one such formalism.

Significant research has been done in the past for developing modeling formalisms for system level design. Process algebras, such as CSP [4] and CCS [8] have been used for verifying distributed software, but have limitations in modeling. For example, CSP allows only rendezvous communication between parallel processes. StateCharts [3] 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 [5]. Formal methods, developed for hardware verification, have been applied to embedded systems like bounded model checking [2] and theorem proving [9]. 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 [9] [1]. A complete methodology for correct digital design has been proposed in [7], but they only consider synchronous models which are insufficient at system level.

The rest of the paper is organized as follows. Section II gives the definition of MA. In Section III we demonstrate how to represent hierarchical system level models in MA. In Section IV we present the formal execution semantics and notions for functional equivalence of models in MA. Section V presents functionality preserving transformations on models and in Section VI we show how these transformations may enable useful refinements during system level design.

## II. Model 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. We define the subset $\mathcal{Q}$ to be the set of all

boolean functions on $\mathcal{V}$.

### A. Ports

Each behavior has an associated object called its interface. The interface carries the data ports of the behavior. In the case of a hierarchical behavior, the ports are identified by association of a variable to the interface. Hence, to sub-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*. 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.

### B. Addressing

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.

### C. Composition Rules

Composition rules on the above objects are defined as relations in MA. Each composition creates a term in MA.

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 $\forall i, 1 \leq i \leq nb, b_i \in \mathcal{B} \cup \mathcal{I}, q \in \mathcal{Q}$. 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 $q$, implying that $b$ must wait for all predecessors.

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 indicates an out-port-map.

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 $\mathcal{I} < p' > \rightarrow b < p >$, In this case, the composition rule indicates an in-port-map.

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 a rendezvous communication mechanism. 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 $c < a >$. Also, any two transactions over the same channel are mutually exclusive in time.

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. We represent a blocking write by the expression $b < p > \mapsto \mathcal{I} < p' >$, where $b < p >$ is the out-port of the writing behavior and $\mathcal{I} < p' >$ is an out-port on the parent of $b$.

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. We represent a blocking read by the expression $< a >: \mathcal{I} < p' > \mapsto b_1 < p_1 > \& b_2 < p_2 > ... \& b_n < p_n >$, where $b_1 < p_1 >$ through $b_n < p_n >$ are the in-port(s) of the receiving behavior(s) and $\mathcal{I} < p' >$ is an in-port on the parent of $b$. The address of the virtual link ($< a >$) will be used for binding this port.

7. Grouping: This composition rule ($R_g$)is used to indicate a collection of compositions. 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 $\forall i, 1 \leq i \leq n$ and $r_i \in \bigcup \{R_c, R_{nw}, R_{nr}, R_t, R_{bw}, R_{br}, R_g\}$.

## III. Model Construction with MA

In this section, we look at how to construct hierarchical system models as expressions in MA. A given hierarchical behavior $b$ has a unique *virtual starting point*(VSP)

and the *virtual terminating point*(VTP). 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 $vsp_b$. 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 $b$.

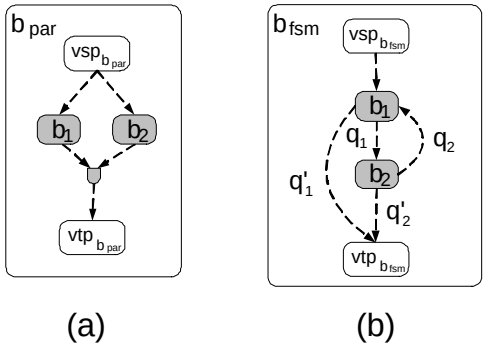


Fig. 1. (a)Parallel and (b)FSM style compositions of behaviors

### A. 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 control relations.

Figure 1(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 1(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
$vsp_{b_{fsm}} \rightsquigarrow b_1.q_1 : b_1 \rightsquigarrow b_2.q_1' : b_1 \rightsquigarrow vtp_{b_{fsm}}$.
$q_2 : b_2 \rightsquigarrow b_1.q_2' : b_2 \rightsquigarrow vtp_{b_{fsm}}$

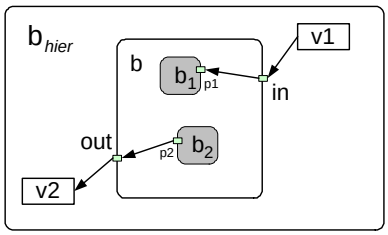


Fig. 2. Using ports for non-blocking data flow

### B. 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 2, 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 use the non-blocking relation $v_1 \rightarrow b < in >$. At the level of $b$, we create a port connection from the interface of $b$ to $b_1$ using the term $\mathcal{I} < in > \rightarrow b_1 < p_1 >$. 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 2. 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 $b_2 < p_2 > \rightarrow \mathcal{I} < out >$.

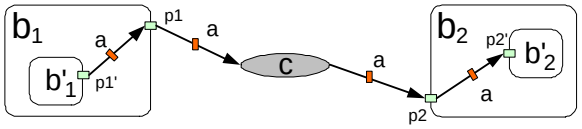


Fig. 3. Blocking data flow bound to channel

### C. Channel Access via Ports

As in the case of non-blocking reads and write, MA provides mechanism for blocking reads and writes via ports. For instance, in Figure 3, 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 $< a >: b_1' < p_1' > \mapsto \mathcal{I} < p_1 >$ 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 $< a >: \mathcal{I} < p_2 > \mapsto b_2' < p_2' >$ 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 $a$.

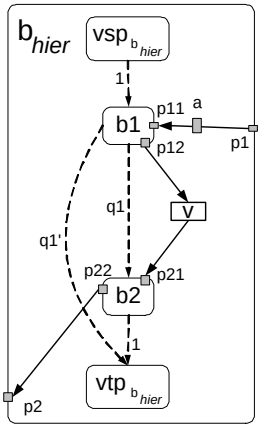


Fig. 4. A hierarchical behavior with local objects and relations

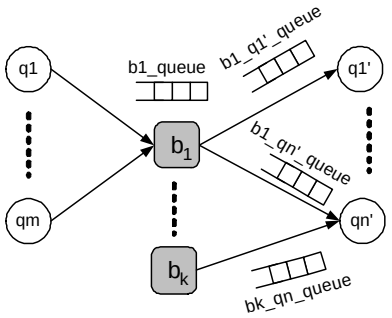


Fig. 5. The firing semantics of BCG nodes

### D. Internal and Interface Terms

Figure 4 shows a hierarchical behavior $b_{hier}$. The expression for the hierarchical behavior is written using compositions involving local objects 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 4, we can see that
$|b_{hier}| =< a >: \mathcal{I} < p_1 > \mapsto b_1 < p_{11} > .$
$b_2 < p_{22} > \rightarrow \mathcal{I} < p_2 >$

Finally, we write the hierarchical behavior as a grouping of its internal and interface terms. Therefore, we get
$b_{hier} = ([b_{hier}].|b_{hier}|)$

## IV. Execution Semantics and Equivalence

Before we attempt to establish equivalence notions for models expressed in MA, we must clearly define the execution semantics of such models. The control dependencies in the model are captured using the *Behavior Control Graph* (BCG), which is similar to the popular computation model of Kahn process network (KPN) [6]. The channels in the model are abstracted away and replaced by corresponding control dependencies, resulting from their rendezvous semantics.

### A. Behavior Control Graph

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

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 5. 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.

### B. Channel Semantics

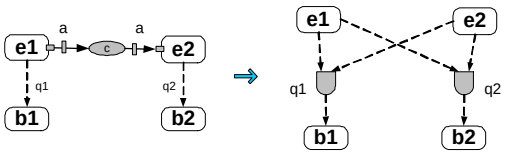


Fig. 6. Resolution of channels into control dependencies

The rendezvous property of a channel would ensure that any behavior following the sender behavior would not execute until the receiver behavior has executed **and**

any behavior following the receiver behavior would not execute until the sender 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 6. 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 $q_2 : e_2 \& e_1 \rightsquigarrow b_2$

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

## V. Functionality Preserving Transformations

In this section, we present some laws of MA that allow us to define functionality preserving model transformations.

### A. Behavior Flattening Laws

Hierarchy is only a modeling artifact, without any influence on functionality. Hence, it may be optimized away. By the semantics of the VSP, any control relation leading to $b_{hier}$ is effectively leading to $vsp_{b_{hier}}$. Similarly, in any control relation where $b_{hier}$ is a predecessor, it may be replaced by $vtp_{b_{hier}}$. Thus, we have

**L 1** $q : x \rightsquigarrow b = q : x \rightsquigarrow vsp_b$

**L 2** $q : b \rightsquigarrow x = q : vtp_b \rightsquigarrow x$

We also have the following laws for port optimization during behavior flattening.

**L 3** $(...y \rightarrow \mathcal{I} < p > ...) < p > \rightarrow x = y \rightarrow x$

**L 4** $x \rightarrow (...\mathcal{I} < p > \rightarrow y...) < p > = x \rightarrow y >$

**L 5** $c < a >: x \mapsto (... < a >: \mathcal{I} < p > \mapsto y...) < p > = c < a >: x \mapsto y >$

**L 6** $c < a >: (... < a >: y \mapsto \mathcal{I} < p > ...) < p > \mapsto x = c < a >: y \mapsto x >$

### B. Identity Elimination Laws

The identity behavior does not modify any variable, so it may be optimized away using appropriate transformations to control and data flow relations in a model.

**L 7** $q_1 : x \rightsquigarrow e . q_2 : e \rightsquigarrow y = q_1 \wedge q_2 : x \rightsquigarrow y$

**L 8** $x < p > \rightarrow v_1 . v_1 \rightarrow e < in > . e < out > \rightarrow v_2 = x < p > \rightarrow v_2$

**L 9** $c < a >: e_1 < out > \mapsto e_2 < in > . e_2 < out > \rightarrow v = e_1 < out > \rightarrow v$

**L 10** $c < a >: e_1 < out > \mapsto e_2 < in > . c' < a' >: e_2 < out > \mapsto e_3 < in > = c < a >: e_1 < out > \mapsto e_3 < in >$

### C. Control Relaxation Laws

Control dependencies between two behaviors, say $x$ and $y$ do not influence the functionality of the model if there is no data dependence between $x$ and $y$. Thus a control dependence $q : x \rightsquigarrow y$ may be removed if $\nexists v$ such that $x < p > \rightarrow v . v \rightarrow y < p' >$ or $q$ depends on $v$. Under these conditions, we can say

**L 11** $R.q : x \rightsquigarrow y = R$

**L 12** $R.q : x_1 \& x_2 \& ... \& x \rightsquigarrow y = R.q : x_1 \& x_2 \& ... \rightsquigarrow y$

## VI. System Level Design and Verification Methodology

In our system level design methodology, the following design steps are encountered as we start from a functional specification model and produce a scheduled bus transaction level model.

1. Behavior partitioning: During behavior partitioning, we choose the number of PEs that will be needed to implement the design and the mapping of leaf behaviors in the specification to those PEs. The output is a parallel composition of hierarchical PE 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 for synchronization amongst behaviors to preserve the original order of execution of the leaf behaviors. The data flow relations are 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, and needs to be routed via identity behaviors using channels. The transformations to derive the output partitioned model use the identity elimination and flattening laws (in either direction).

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. Reordering of behaviors can also take place as a result of communication scheduling. Essentially, we can use the transformations allowed by control relaxation laws (in either direction) to create a new schedule statically.

3. RTOS insertion: PEs that implement software may provide for dynamic scheduling. In this case, the nondeterminism of concurrency is resolved at execution time. The ordering of parallel behaviors 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).Therefore, for functionally correct implementation of 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.

4. Bus protocol 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. Transactions on a channel are mutually exclusive and the new arbiter must satisfy this property. For a correct implementation of arbitration policy, we have the same scenario as that in RTOS insertion. We can use property checking to verify that the arbitration policy preserves the functionality of the model. If we can prove that the arbiter behavior will never cause a deadlock **and** will eventually schedule a transaction request then we can abstract it away.

## VII. Conclusions

In this paper, we introduced a formalism called Model Algebra, which is employed in deriving detailed system level models from more abstract ones using correct transformations. We established a notion of functional equivalence and refinement using partial order traces. The axioms and rewriting rules of MA were then shown to be sound, which led us to defining functionality preserving model transformations. Finally, we presented a system synthesis scenario and showed how our formalism can be used in a system design methodology. The complexity of modern systems stresses upon the need for such a formal approach, so as to avoid the overhead of verifying and debugging all models during design space exploration.

## References


[1] 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.

[2] 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.

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

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

[5] 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.

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

[7] 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.

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

[9] 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.