Transaction Routing and its Verification by Correct Model Transformations

Samar Abdi and Daniel Gajski
Center for Embedded Computer Systems
University of California
Irvine, CA 92697-2625, USA
Email: sabdi, gajski@uci.edu

***Abstract*— This paper presents model transformations that are encountered in refining an abstract point-to-point transaction between two processes into a complex transaction, routed over the communication architecture, consisting of multiple busses and bridges. These transformations form part of synthesizing an abstract specification model into a detailed model representing the implementation of that specification onto a platform. We present these transformations in the context of a modeling formalism that has well defined execution semantics and a notion of functional equivalence. The transformations are proven correct using our notion of equivalence. We also present methods for deriving the proof of equivalence between the abstract model and the refined model . Based on these methods, we have implemented a tool that automatically proves whether or not the model generated after transaction routing is indeed equivalent to the input model. Experimental results for large industrial examples demonstrate the feasibility, utility and efficiency of our tool and the underlying methods.**

I. Introduction

Transaction level (TL) design is being adopted to combat the rising complexity of modern embedded designs. Design methodologies now involve several modeling stages and platform/application updates before a cycle accurate implementation is considered. At each step, the TL model (TLM) is transformed to reflect the design decision made at that step. However, it is imperative that the functionality of the model is preserved as the design progresses through these incremental refinements. In this paper, we present a technique for functional verification of model refinement resulting from transaction routing.

A possible design methodology is shown in Figure 1. We start by creating an executable model of the system that captures the application as a high level TLM consisting of concurrent processes communicating with abstract point-to-point channels. The communication architecture is described as a netlist of processing elements (PEs) and busses. During TLM refinement, design decisions are used to map the application processes to the PEs in the platform. The point to point channels are routed over communication paths consisting of busses and bridges. The resulting model is a detailed low level TLM where the abstract point to point channel communication is now routed as communication over shared bus channels and bridge processes. The verification problem we wish to address is to prove or disprove the functional equivalence of high TLM and low TLM.

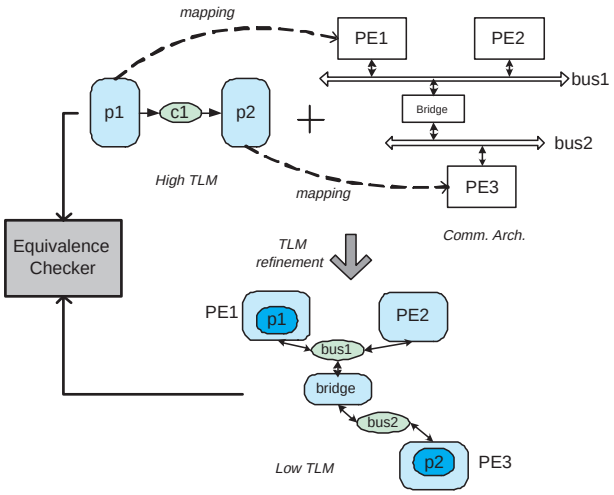

Fig. 1. TLM synthesis methodology

In order to solve this verification problem, we define a formal representation of TLMs. Then we define functionality transformation laws on these formal representations. Using these laws, we represent the refienement as a sequence of correct transformations. Based on this theory, a verification tool takes the low TLM and automatically applies a sequence of correct transformations on it to derive the high TLM, to prove their functional equivalence.

There is a huge body of research in functional equivalence verification of high level system models, mostly from the software community. Symbolic simulation has been used in [8] to verify equivalence of terminating embedded software. In [11], the authors use textual comparisons of models to check consistency. Checking of C models against their Verilog implementations has been proposed in [7] using bounded model checking. Correct-by-construction techniques have been implemented for system design, notable in ForSyde [12] tool set. The need for high level modeling of embedded systems has given rise to system level design languages (SLDLs) such as like SystemC 2.0 [1] and SpecC [9]. This has led to research being directed towards modeling and verification at system level in order to verify the correctness of design steps. Traditional software model checking and bounded model checking [6] allow property verification of high level models

written in C-like languages. However there has been little work in refinement verification of system level models using model transformations.

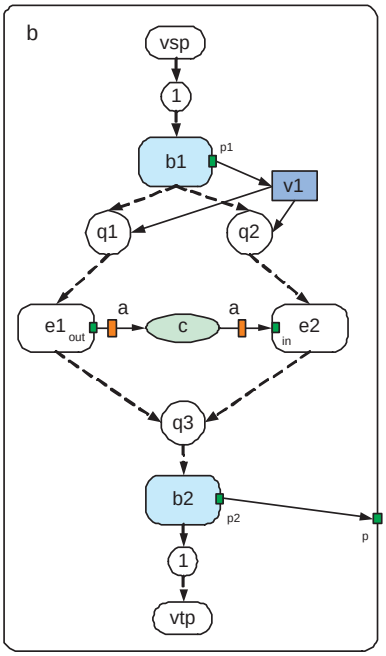

Fig. 2. Model created using MA

II. Model Algebra

We define the formalism of Model Algebra (MA) to represent TLMs in a concise way and to reason about their composition and transformations. In this section, we will discuss MA objects and composition rules, model execution semantics and functional equivalence notion.

A. Definition and Model Creation

Figure 2 shows a model created using the objects and composition rules of MA. The objects are behaviors (round-edge boxes) to caputure computation, variable (rectangles) and channels (ellipses) to capture communication, control conditions (circles), behavior interfaces and ports (on behavior interface). Unshaded round-edge boxes represent identity behaviors that simply copy inputs to outputs. Control dependencies are represented using broken arrows. For example, broken edges from b_1 to q_1 and q_1 to e_1 imply that e_1 executes after b_1 has executed and condition in node q_1 has evaluated to true. In MA, we write this succinctly as $q_1 : b_1 \rightsquigarrow e_1$. Complex control dependencies may be created like the one at node q_3. Here, behavior b_2 executes only after both e_1 and e_2 have finished and condition in q_3 evaluates to true. This is represented in MA using term $q_3 : e_1 \& e_2 \rightsquigarrow b_2$. Data dependencies are represented by connections from ports to variables. For example, the edge from port p_1 of behavior b_1 to variable v_1 implies that during execution, behavior b_1 writes to v_1. This is written in MA as $b_1 < p_1 > \rightarrow v$. Several transactions may be sent over the same channel and are distinguished by address labels. For instance, a transaction from e_1 to e_2 over channel c with address a is shown in Figure 2. In MA, we will write this transaction as the term $c < a > : e_1 < out > \mapsto e_2 < in >$. MA allows hierarchy using the interface object (I) and port mapping. For example, a hierarchical behavior b has port p that is mapped to port p_2 of behavior b_2. This port mapping is written in MA as the term $b_2 < p_2 > \rightarrow I < p >$. The association of p with I in this term implies that p is a port of the parent behavior. Each hierarchical behavior also has two identity behaviors that represent its unique virtual starting point (vsp) and virtual termination point (vtp). Hierarchical behaviors in MA are expressed as a grouping of all the sub-behaviors and the local terms, incuding control and data dependencies and port mappings. Hence, in MA, we can write b completely as
$b = [vsp].[vtp].[b_1].[b_2].1 : vsp \rightsquigarrow b_1.q_1 : b_1 \rightsquigarrow e_1.$
$q_2 : b_1 \rightsquigarrow e_2.b_1 < p_1 > \rightarrow v_1.v_1 \rightarrow q_1.v_1 \rightarrow q_2.$
$c < a > : e_1 < out > \mapsto e_2 < in > .q_3 : e_1 \& e_2 \rightsquigarrow b_2.$
$b_2 < p2 > \rightarrow I < p > .1 : b_2 \rightsquigarrow vtp$

B. Execution Semantics of MA Models

The control dependency relations of MA create execution dependencies between behaviors. Channel transactions have double handshake semantics, which means that the receiver blocks until the sender sends the data and the sender blocks its following behaviors until the receiver has received the data. Therefore, channel transactions also create execution dependencies between behaviors. We further define the domintaor relation as follows. If a behavior b always executes once before every unique execution of b', we say that b *dominates* b' and write this relation as $b \triangleright b'$.

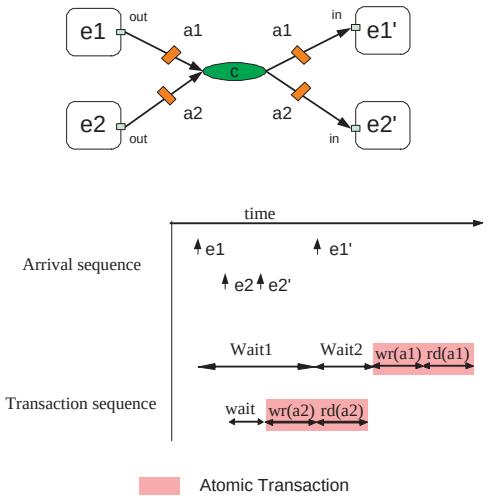

Fig. 3. Multiple competing transactions on a single channel

1) Channel Semantics: Consider the configuration shown in figure 3. In this case, two transaction links, addressed a_1 and a_2, are shared over channel c. These links can be written in MA as
$c < a_1 > : e_1 < out > \mapsto e_1' < in >.$
$c < a_2 > : e_2 < out > \mapsto e_2' < in >$
The sequence 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 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 a_2 is in progress.

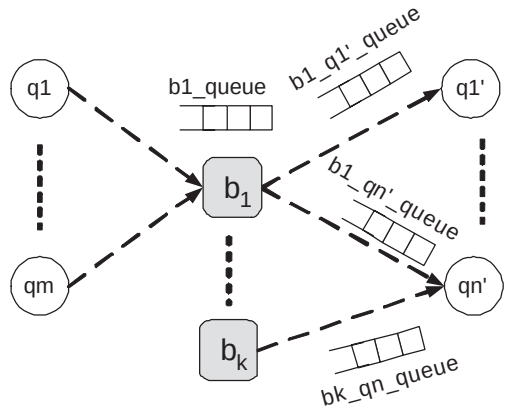

Fig. 4. The firing semantics of BCG nodes

2) Model Semantics: In order to define the execution semantics of MA, the control relations in the model are captured using the *Behavior control graph*(BCG) as shown in Figure 4. It is a directed graph with two types of nodes, namely *behavior nodes* and *control nodes*. The behavior nodes, as the name suggests, indicate behavior execution, while the control nodes represent evaluation of control conditions that lead to further behavior executions. A MA 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 $((b_1,q),...,(b_n,q),(q,b))$. Each behavior node in the BCG has one associated queue, for instance $b1\_queue$ for behavior node b_1. All incoming edges to a behavior node represent the various writers to the behavior's queue. The control node (shown by circular node), on the other hand, has as many queues as the number of incoming control edges. Note that a control node has only one outgoing edge as per the control flow composition rule of MA.

A behavior node blocks on an empty queue and executes if there is at least one token in its queue. Upon execution, a behavior node first updates all variables that it is writing to and one token is dequeued from the node's queue. Then, it 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. A control node, sequentially checks all its queues and blocks on empty queues. That is if an incoming edge's queue is empty, we must wait until the behavior on the other end of the edge has executed. If the queue is not empty, it dequeues a token from the queue and proceeds to check the next queue. After one token from each of the incoming queues is dequeued, the condition inside the control node is evaluated. If the result is TRUE, then the control node enqueues one token to the queue of the destination behavior.

C. Equivalence Notion for MA Models

Our notion of functional equivalence is based on the trace of values that certain interesting variables hold during model execution. We will refer to such variables as *observed variables*. Given a model M and a set of its observed variables, say OV_M, let $Init_M$ be the initial assignment of all the variables in OV_M. Let $v \in OV_M$. We define $\tau_M(v, Init_M)$ as the set of all possible traces of values assumed by v, when model M is executed with initialization $Init_M$.

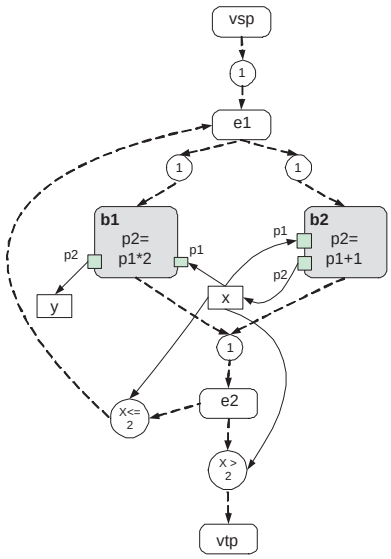

Fig. 5. Example of a model (M) with several possible value traces

Figure 5 shows a model M with behaviors b_1 and b_2 composed in parallel. Let x and y be the observed variables, with an initial assignment of 0 and U (undefined), respectively. When model M is executed, either b_1 or b_2 may execute first after e_1. Therefore, the value of y may become 0 if b_1 executes first, or become 2 if b_2 executes first (modifying x to 1). However, by the time e_2 executes x is definitely 1. Since $x < 2$, e_1 executes again, allowing either b_1 or b_2 to execute first. The process continues until x becomes 3, Hence, for the given scenario, we have

$$\begin{aligned} OV_m &= \{x,y\}, \\ Init_M &= \{0,U\}, \\ \tau_M(x, Init_M) &= \{\{0,1,2,3\}\} \\ \tau_M(y, Init_M) &= \{\{U,0,2,4\},\{\{U,0,2,6\},\{\{U,0,4\}, \\ &\quad \{\{U,0,4,6\},\{\{U,2,4\},\{\{U,2,6\}, \\ &\quad \{\{U,2,4,6\}\} \end{aligned}$$

Given two models M and M' expressed using model algebra, we wish to determine if they are equivalent with respect to some well defined notion of equivalence. First, we need to

determine a correlation between the two models based on their respective observed variables. Informally speaking, we consider two models to be functionally equivalent, if they have one-to-one correspondence of observed variables and the trace of values assumed by those variables during model execution is identical for all identical initial assignments.

Formally, in order to show equivalence of M and M', we require that $|OV_M| = |OV_{M'}|$ and there exists a bijective mapping from OV_M to $OV_{M'}$. We will represent the mapping of respective elements by the $\leftrightarrow$ symbol. Therefore, we have $\forall v \in OV_M, \exists v' \in OV_{M'}$, such that $v \leftrightarrow v'$.
Let $Init_M$ be the initial assignment of all variables in OV_M and $Init_{M'}$ be the initial assignment of all variables in $OV_{M'}$. If $\forall Init_M, Init_{M'}, \forall v \in OV_M, \exists v' \in OV_{M'}, v \leftrightarrow v'$, such that initial assignment of v is equal to the initial assignment of v' and $\tau_M(v, Init_M) = \tau_{M'}(v', Init_{M'})$, then M is said to be functionally equivalent to M'.

III. Transformation Rules

In this section we present the transformation laws of MA that are based on the execution semantics of MA models and notion of their functional equivalence. The transformation laws define how a model expressed in MA may be syntactically manipulated while preserving functional equivalence as per the notion defined above. Here we will give a sketch of the soundness proofs for each rule. More details of the proofs are presented in [2].

The transformation rules presented here are applicable to verification of not only transaction routing, but also various other refinements such as behavior partitioning [3], [4] and scheduling [5]. Although some refinement require splitting or merging of behaviors, none of the rules allow splitting or merging of leaf level behaviors. Such refinements can still be proven by careful modeling of the original models. For instance, if the original model has a behavior that will be eventually split, it must be described as a hierarchical composition of the split behaviors. Similarly, instead of combining two leaf behaviors into a bigger leaf behaviors, all leaf behavior combinations should be described as hierarchical behaviors. Using the notion of hierarchy, we can bypass the problem of behavior splitting or merging.

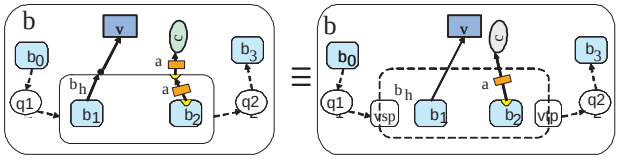

Fig. 6. Flattening transformation rule

A. Flattening of Hierarchical Behaviors

Figure 6 shows the flattening rule on behavior b_h. Any control relation leading to b_h is replaced by one leading to vsp, where vsp is the virtual starting point of behavior b_h. Similarly, any control relation from b_h is replaced by comtrol relation from vtp. Variable writes and channel transactions from b_h are replaced as per the corresponding port mapping inside b_h. Similarly, all port maps at scope of b involving ports of b_h are replaced by corresponding port maps of b_h's sub-behaviors.

The soundness of flattening rule follows from the definition of hierarchy in MA. By definition of the VSP and VTP, $q : x_1 \& x_2 ... \& x_n \rightsquigarrow b_h = q : x_1 \& x_2 ... \& x_n \rightsquigarrow vsp_h$, and $q : ... \& b_h \& ... \rightsquigarrow x = q : ... \& b_h \& ... \rightsquigarrow vtp_h$.
For all b' such that b' is a sub-behavior of b_h, the execution of b' updates all variables mapped to its outport. If the out ports of b' are mapped to out ports of b_h, then the variables mapped to the latter will be updated. Therefore, by inductive reasoning updating port maps during flattening is sound. Similar reasoning can be used for port mappings for ports connected to channels.

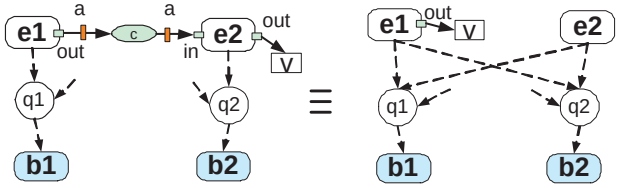

Fig. 7. Link resolution rule

B. Control flow resolution of links

Figure 7 illustrates control dependency extraction and modification of data dependencies resulting from link resolution. On the RHS, the link is replaced by control dependencies from the sender to all behaviors following the receiver. Similarly, we have new control dependencies from receiver to all behaviors following the sender. The variable written by the receiver identity is now written by the sender.

On the LHS, transaction $c < a >: e_1 < out > \mapsto e_2 < in >$ implies that execution of e_1 is not complete until execution of e_2 is complete and vice versa. In other words, all control nodes that have edges from e_1 will not be evaluated until e_2 has executed. This is equivalent to adding edge (e_2, q_1). Similar reasoning applies for adding edge (e_1, q_2). By identity definition, when e_1 executes, it reads its in port and sends data to e_2 via c using address a. Since e_2 writes to variable v, the channel transaction and relation $e_2 < out > \rightarrow v$ is equivalent to $e_1 \rightarrow v$. Hence, link resolution is sound.

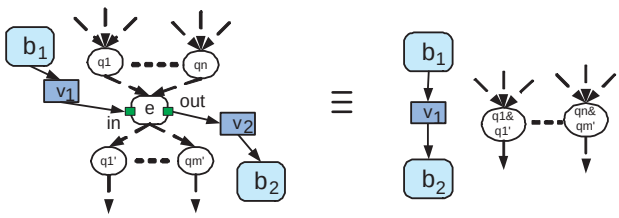

Fig. 9. Identity elimination rule

C. Identity elimination

The identity elimination (IE) rule is shown in Figure 9. Here, we have n incoming control edges and m outgoing

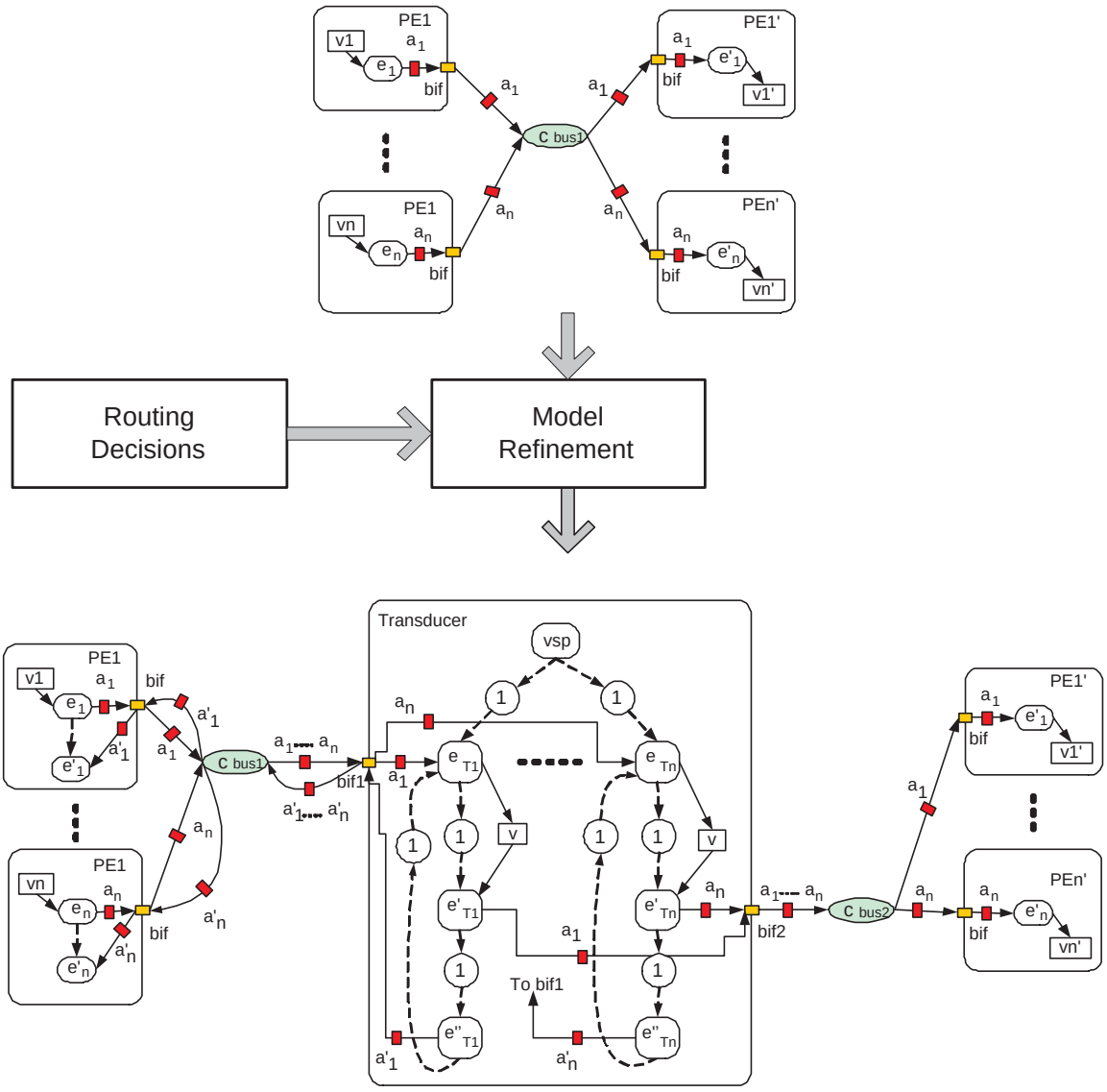

Fig. 8. Routing transactions via transducer

control edges. After the transformation, on RHS we get $m \times n$ merged nodes, where each node has the control condition as the conjunction of the respective incoming and outgoing control condition. Since v_2 is a copy of v_1, b_2 may read v_1 instead of v_2.

Consider control node q_i, where $1 \leq i \leq n$ and control node q'_j, where $1 \leq j \leq m$, such that edges (q_i, e) and (e, q'_j) exist in the model on the LHS in Figure 9. Let b'_j be a behavior node such that edge (q'_j, b'_j) also exists in the LHS model. Let b_{i1} through b_{ik} be all the behaviors with edge to q_i. Without loss of generality, we assume that during model execution, b_{i1} through b_{ik} execute resulting in the evaluation of q_i. If q_i is true, e executes leading to evaluation of q'_j. If q'_j also evaluates to true, then b'_j will also execute. Hence, we deduce that b'_j will execute if b_{i1} through b_{ik} execute and $q_i \wedge q'_j$ evaluates to true. Generalizing our result for all $i, 1 \leq i \leq n$ and for all $j, 1 \leq j \leq m$, we deduce that the execution results will be identical for models on LHS and RHS. Therefore, IE rule is sound.

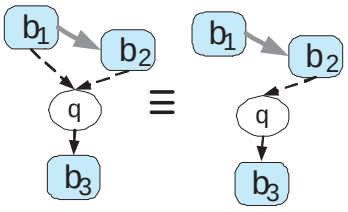

Fig. 10. RCDE rule

D. Redundant control dependency elimination

If a complex control flow relation has more than one predecessors, where one predecessor dominates another, then

the relation can be simplified using redundant control dependency elimination (RCDE) rule as shown in Figure 10. The dominator relation between b_1 and b_2 ($b_1 \triangleright b_2$)is shown by a solid grey arrow from b_1 to b_2. The model on RHS of is derived by removing the control edge from b_1 to q. Given $b_1 \triangleright b_2$, the control node q will evaluate only if both b_1 and b_2 have executed. Now, by the dominator definition, we know that any execution of b_2 implies that b_1 must have executed earlier therefore an edge from b_1 to q is redundant. Thus, RCDE rule is sound.

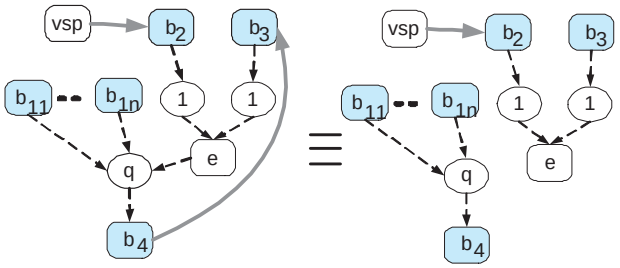

Fig. 11. Streamlining rule

E. Streamlining

As illustrated in Figure 11, streamlining rule can be applied for a given control node q with successor b_4 and $n+1$ predecessors, where one of the predecessors is an identity behavior e. Identity e, in turn has two control dependencies (both with condition 1), from two behaviors, say b_2 and b_3. Behavior b_2 is either *VSP* of the model or dominated by *VSP*. Behavior b_3 is either b_4 or dominated by b_4. If $b_3 \neq b_4$, then b_3 must execute at least once *between* any two executions of b_4. The model is transformed by removing the control dependency from e to b_4, thus deleting the edge from e to q.

On the LHS, we have a control node q with control edges from behaviors b_{11} through b_{1n} and identity e. Also, q has a control edge to behavior b_4. Now, from the execution semantics b_4 will execute if all of b_{11} through b_{1n} and identity e execute and then condition q evaluates to true. Node e has two incoming control paths from b_2 and b_3. Therefore, e will execute if and only if either b_1 of b_2 execute. Since b_1 is either *vsp* or dominated by it, b_2 will execute only once. Also, b_3 is either same as b_4 or executes once for each execution of b_4 (after b_4 has executed). Therefore, the first execution of e will result from the control path from b_2, leading to a potential execution of b_4, if all of b_{11} through b_{1n} execute. If b_4 executes, then b_3 will also execute, leading to another execution of e and a subsequent potential execution of b_4 if all of b_{11} through b_{1n} execute. Therefore, every execution of b_4 is dependent only on the execution of b_{11} through b_{1n} and the evaluation of q to be true. Thus, the control dependency represented by edge (e,q) is redundant. Hence, streamlining rule is sound.

IV. Transaction Routing

During transaction routing, the point to point channels are routed over new bus channels and bridges. If the sender and receiver PEs of a point to point transaction are connected to the same bus, then the transaction address label can simply be transferred from the point to point channel to the bus channel. Since channels are simply containers for transaction address labels, changing the location of label from one channel to another is functionality preserving. The more complicated case is one where a channel transaction is split into several transactions over multiple busses and bridges. In this section, we will present the refinement algorithm for manipulating the MA representation for such synthesis decisions. For the refinement, we also present a proof of correctness using the aforementioned MA transformation rules.

A. Refinement Algorithm

Figure 8 shows a TL refinement for transaction routing. On the LHS, we have PEs connected with point to point channels. On the RHS, the transactions are routed using bus channels c_{bus1} and c_{bus2} over a bridge called transducer. We now present the general algorithm for performing the model refinement in MA for such transaction routing. The algorithm can easily be generalized for inserting several transducers in a multi-hop transaction.

Let there be a model M of a system with a bus represented by channel c. Let there be n transactions over c from and to different PEs connected to c. Suppose we change the communication architecture such that the communicating PEs are now connected to two different busses, represented by channels c_1 and c_2. The original transaction links represented by addresses a_1 through a_n must now go over two busses c_1 and c_2, via a new component called the *Transducer*. The transducer has two interfaces bif_1 and bif_2 for the two busses c_1 and c_2, respectively. Inside the transducer, we have a parallel composition of n sets of 3 identity behaviors, each set responsible for routing of transactions on one original link. A set consists of a receiver identity behavior that copies the incoming data from one bus into a local buffer, and a sender identity behavior that sends the data from buffer over the other bus to the intended recipient and a notification identity behavior that sends a notification transaction to the sender after the sender has executed. Algorithm 1 gives pseudo code for the refinement resulting from transducer insertion.

B. Proof of Correctness

Figure 12 shows the basic proof steps for showing equivalence between models before and after transducer insertion. Without loss of generality, we assume a transducer between two channels c_1 and c_2 as shown in the flattened model in Figure 12(a). The transducer replaces a direct channel transaction between identities e_1 and e_2. The transducer consists of an endless loop with a sequential body of 3 identities namely e_T, e'_T and e''_T. Identity e_T has a transaction link from e_1. After this transaction occurs, e_T copies the received data in a local variable v. Thus v is a copy of v_1. Then, e'_T executes, sending data from v to e_2 over channel c_2. Finally, after this transaction is complete, identity e''_T executes and synchronizes with e'_1

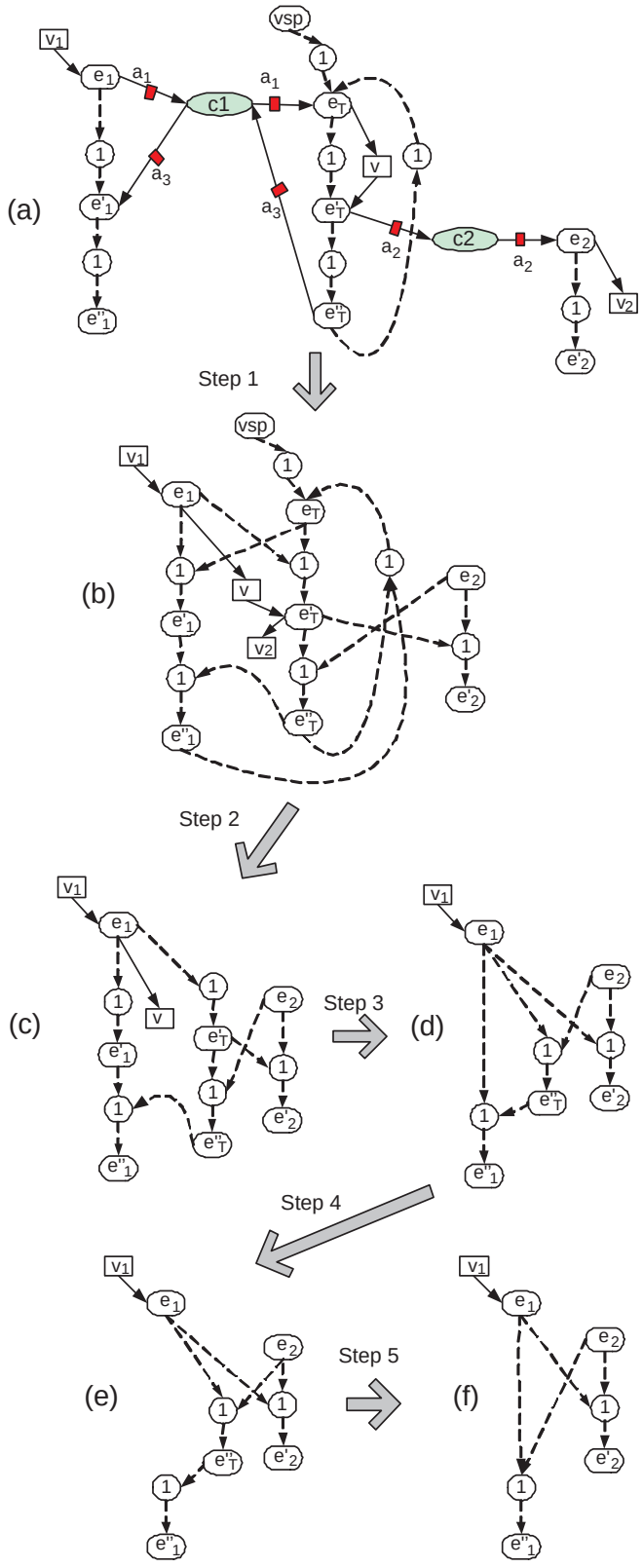

Fig. 12. Proof steps for transducer insertion during transaction routing refinement

Algorithm 1 Insert Transducer

1: $M = M.[T].1 : vsp \rightsquigarrow T$
2: **for** $i = 1to$ TO n **do**
Require: $\exists PE_s, PE_r \in B$, s.t.
$c < a_i >: PE_s < bif >\mapsto PR_r < bif > \in M$
3: **if** PE_s ConnectedTo $bus1 \wedge PE_r$ ConnectedTo $bus2$ **then**
4: $M = (M - c < a_i >: PE_s < bif >\mapsto PR_r < bif >).c_1 < a_i^1 >: PE_s < bif >\mapsto T < bif_1 >$
5: $M = M.c_2 < a_i^2 >: T < bif_2 >\mapsto PR_r < bif >$
6: $T = T.1 : vsp_T \rightsquigarrow e_{Ti}.1 : e_{Ti} \rightsquigarrow e'_{Ti}.1 : e'_{Ti} \rightsquigarrow e''_{Ti}.1 : e''_{Ti} \rightsquigarrow e_{Ti}$
7: $T = T.e_{Ti} < out >\rightarrow v_i.v_i \rightarrow e'_{Ti} < in >$
8: $T = T.a_i^1 : I < bif_1 >\mapsto e_{Ti} < in > .a_i^2 : e'_{Ti} < out >\mapsto I < bif_2 > .a_i^3 : e''_{Ti} < out >\mapsto I < bif_1 >$
9: **else**
10: $M = (M - c < a_i >: PE_s < bif >\mapsto PR_r < bif >).c_2 < a_i^1 >: PE_s < bif >\mapsto T < bif_2 >$
11: $M = M.c_1 < a_i^2 >: T < bif_1 >\mapsto PR_r < bif >$
12: $T = T.1 : vsp_T \rightsquigarrow e_{Ti}.1 : e_{Ti} \rightsquigarrow e'_{Ti}.1 : e'_{Ti} \rightsquigarrow e_{Ti}$
13: $T = T.e_{Ti} < out >\rightarrow v_i.v_i \rightarrow e'_{Ti} < in >$
14: $T = T.a_i^2 : I < bif_2 >\mapsto e_{Ti} < in > .a_i^1 : e'_{Ti} < out >\mapsto I < bif_1 >$
15: **end if**
16: **end for**

We start by resolving all the transaction links in the model into appropriate control and data dependency relations using the link resolution transformation rule (Step 1). By the application of this rule, we replace transaction and data relations
$c_1 < a_1 >: e_1 < out >\mapsto e_T < in > .e_T < out >\rightarrow v$
with $1 : e_1 \& e_T \rightsquigarrow e'_T.1 : e_1 \& e_T \rightsquigarrow e'_1.e_1 < out >\rightarrow v$
Transaction relations between e'_T and e_2 and data relations of e_2 as follows
$c_2 < a_2 >: e'_T < out >\mapsto e_2 < in > .e_2 < out >\rightarrow v_2$
are replaced by control and data relations
$1 : e_2 \& e'_T \rightsquigarrow e''_T.1 : e_2 \& e'_T \rightsquigarrow e'_2.e_2 < out >\rightarrow v_2$ The notification transaction between e'_1 and e''_T is also replaced similarly using the link resolution rule. Figure 12(b) ahows the model after transformations of Step 1.

In Step 2, we apply the streamlining rule in the same fashion as used in the proof for RPC style behavior partitioning algorithm. The rule eliminated control dependencies from e_T to e'_1 and from e_T to e'_t. As a result, e_T does not have anything executing after it and it does not modify and variable in the model, so it is removed from the model along with all its control and data relation. The model after Step 2 is shown in Figure 12(c).

In Step 3, we optimize away identities e'_T and e'_1 using the identity elimination rule. In Step 4, we use the RCDE rule to remove the redundant control dependency from e_1 to e''_1. This is because the control dependency is cause by the relation $1 : e_1 \& e''_T \rightsquigarrow e''_1$ and $e_1 \triangleright e''_T$, thereby allowing application of RCDE. Finally, in Step 5, we optimize away identity e''_T using the identity elimination rule. The final resulting model, as shown in Figure 12(f), is identical to the one derived after link

resolution of a direct link from e_1 to e_2. Therefore, we have proved that the transducer insertion algorithm is functionality preserving.

Application	High TLM	Low TLM	# Trans.	Verification Time
Voice Codec	B:188, D:138, Q:202; CD:428, DD:453	B:204, D:144, Q:221; CD:463, DD:470	Total:7113 (Flat:4726, IE:1600, LR:727, RCDE:18, Str:42)	3.7 sec
MP3 Decoder	B:92, D:76, Q:132; CD:197, DD:160	B:98, D:88, Q:150; CD:224, DD:192	Total:3787 (Flat:2520, IE:848, LR:371, RCDE:12, Str:36)	2.2 sec

Fig. 13. Performance of verification tool

V. Experimental Results

A verification tool based on the proof technique in Section IV-B, was developed in C++ for verifying transaction routing refinement of SpecC models. We present here, results from two applications namely, a GSM voice codec application [10] and a MP3 decoder. The table in Figure 13 shows results for the verification of transaction routing on the two applications. An abstraction module was used to derive the MA representation from a SpecC Model. The MA representation was stored as a graph with three types of nodes, namely behavior (B), data variable (D) and control condition (Q). Control dependencies (CD) and data dependencies (DD) were stored as graph edges. The size of the graphs for the high and low TLMs in terms of these nodes and edges is given for the two benchmarks.

The transformation rules in Section III were then used to derive the graph for high TLM from graph for low TLM. The derived graph was then compared to the original high TLM graph using a simple graph isomorphism checker. The total number of transformation rules applied are given in the column (#Trans). Along with total number, we provide the number of individual rule applications namely for Flattening (Flat), Identity Elimination (IE), Link Resolution (LR), Redundant Control Dependency Elimination (RCDE) and Streamlining (Str). The order of rule applications corresponds to the proof steps, that is the order is predetermined. The verification time is the total measured CPU time on a 2 GHz Pentium machine under Linux. It consists of time taken for SpecC to MA conversion, various rule applications and isomorphism checking for both the high and low TLMs. To test for negative results, faults were injected in the low TLM by removing certain links. This lead to mismatch of the transformed graphs. The results presented here are representative of several routing decisions that were performed for the two benchmarks. All transaction routing refinements were verified in the order of few seconds, thereby making our verification approach highly attractive.

VI. Conclusion

We presented a technique to check the functional equivalence of high and low level TLMs, before and after transaction routing refinements. There are two unique advantages of our approach. First, we define a mathematical foundation for representing and reasoning about TLMs. Second, the well defined semantics of high and low TLMs and proof techniques for checking their equivalence greatly reduce verification time. As a result, the designer does not need to perform costly simulations after every modification to the design implementation. Our experimental results demonstrated the practical feasibility of our verification technique on industrial examples. In the future, we would like to extend the objects and rules of MA to verify more complex system level refinements. For example, the transducer implementation may have a shared protected buffer for temporarily storing different transactions. We will need to apply different rules that can demonstrate the equivalence between shared buffer and separate buffer implementations of the transducer. These and other useful refinements are the topic of our continuing research.

Acknowledgments

The authors would like to thank the various reviewers for their contributions in improving the quality of this paper.

References

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

[2] S. Abdi. Functional Verification of System Level Model Refinements, PhD Thesis. December 2005.

[3] S. Abdi, D. Gajski. Model Validation for Mapping Specification Behaviors to Processing Elements. In *Proceedings of IEEE International High Level Design Validation and Test Workshop*, November 2004, Sonoma, CA, USA.

[4] S. Abdi, D. Gajski. Automatic Generation of Equivalent Architecture Model from Functional Specification. In *Proceedings of ACM Design Automation Conference*, June 2004, San Diego, CA, USA.

[5] S. Abdi, D. Gajski. Functional Validation of System Level Static Scheduling. In *Proceedings of Design Automation and Test in Europe*, March 2005, Munich, Germany.

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

[7] E. Clarke, D. Kroening, and K. Yorav. Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking. In *DAC '03: Proceedings of the 40th Design Automation Conference*, pages 368–371, New York, NY, USA, 2003. ACM Press.

[8] X. Feng and A. J. Hu. Cutpoints For Formal Equivalence Verification of Embedded Software. In *EMSOFT '05: Proceedings of the 5th ACM International Conference on Embedded Software*, pages 307–316, New York, NY, USA, 2005. ACM Press.

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

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

[11] 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.

[12] I. Sander, A. Jantsch, and Z. Lu. Development and Application of Design Transformations in FORSYDE. In *DATE '03: Proceedings of Design, Automation and Test in Europe*, page 10364, Washington, DC, USA, 2003. IEEE Computer Society.