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.

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 refinement 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.

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 capture 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 \( q_1 : b_1 \sim e_1 \). 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 \( q_3 : e_1 & e_2 \sim b_2 \). 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 \( b_1 < p_1 \rightarrow v_1 \). 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 \( c < a > : e_1 < out > \rightarrow e_2 < in > \). 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, including control and data dependencies and port mappings. Hence, in MA, we can write \( b \) completely as \( b = [vsp], [vtp], [b_1], [b_2], 1 : vsp \sim b_1, q_1 : b_1 \sim e_1, q_2 : b_1 \sim e_2, b_1 < p_1 \rightarrow v_1, v_1 \rightarrow q_1, q_1 \rightarrow q_2, c < a > : e_1 < out > \rightarrow e_2 < in >, q_3 : e_1 & e_2 \sim b_2, b_2 < p_2 \rightarrow I < p >, 1 : b_2 \sim vtp \).

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 domintor 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 \( b \triangleright b' \).

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

- \( c < a_1 > : e_1 < out > \rightarrow e_1' < in > \)
- \( c < a_2 > : e_2 < out > \rightarrow e_2' < in > \)

The sequence diagram shows the actual arrival schedule of

![Figure 2. Model created using MA](image)

![Figure 3. Multiple competing transactions on a single channel](image)
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 \( \text{wait}_2 \) since the transaction \( a_2 \) is in progress.

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 to 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 \textit{observed variables}. Given a model \( M \) and a set of its observed variables, say \( OV_M \), let \( \text{Init}_M \) be the initial assignment of all the variables in \( OV_M \). Let \( v \in OV_M \). We define \( \tau_M(v,\text{Init}_M) \) as the set of all possible traces of values assumed by \( v \), when model \( M \) is executed with initialization \( \text{Init}_M \).

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

\[
OV_M = \{x,y\},
\]
\[
\text{Init}_M = \{0,U\},
\]
\[
\tau_M(x,\text{Init}_M) = \{0,1,2,3\}
\]
\[
\tau_M(y,\text{Init}_M) = \{0,2,4,6\}, \{0,2,6\}, \{0,4\}, \{0,4,6\}, \{2,4\}, \{2,6\}, \{2,4,6\}
\]

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 initial identical 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.

#### 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 control 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, $q : x_1 & x_2 & \ldots & x_n \leadsto b_h = q : x_1 & x_2 & \ldots & x_n \leadsto vsp_h$, and $q : \ldots & b_h & \ldots \leadsto x = q : \ldots & b_h & \ldots \leadsto vtp_h$.

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.

#### 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 $c < a >: e_1 < out > \rightarrow e_2 < in >$ 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 $e_2 < out > v$ is equivalent to $e_1 \rightarrow v$. Hence, link resolution is sound.

#### C. Identity elimination

The identity elimination (IE) rule is shown in Figure 9. Here, we have $n$ incoming control edges and $m$ outgoing
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 $q_i \land q'_j$ 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.

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.

![Streamlining rule](image)

**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 $b_1$. The model is transformed by removing the control dependency from $e$ to $b_3$, 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 $b_1$ or $b_2$ 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 $b_3$, 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 $e_T''$ over channel $c_2$. Finally, after this transaction is complete, identity $e_T''$ executes and synchronizes with $e'_1$. 


Algorithm 1 Insert Transducer
1: $M = M \cup \{T\}, 1: vsp \sim T$
2: \textbf{for} $i = 1$ TO $n$ \textbf{do}
3: \textbf{if} $PE_i, PE_r \in B, \text{ s.t.}$
4: \hspace{1em} $c < a_i >: PE_i < \text{bif} > \Rightarrow PR_i < \text{bif}> \in M$
5: \hspace{2em} $M = (M - c < a_i >: PE_i < \text{bif} > \Rightarrow PR_i < \text{bif} > ) . c_i < a_i >: PE_i < \text{bif} > \Rightarrow T < \text{bif}_1 >$
6: \hspace{2em} $M = M . c_i < a_i >: T < \text{bif}_2 > \Rightarrow PR_i < \text{bif} >$
7: \hspace{2em} $T = T . e_i < \text{v}_i > \Rightarrow v_i . v_i \sim e_i < \in >$
8: \hspace{2em} $T = T . a_i < \text{out} > \Rightarrow e_i < \text{out} > \Rightarrow I < \text{bif}_1 >$
9: \hspace{1em} \textbf{else}
10: \hspace{2em} $M = (M - c < a_i >: PE_i < \text{bif} > \Rightarrow PR_i < \text{bif} > ) . c_2 < a_i >: PE_i < \text{bif} > \Rightarrow T < \text{bif}_2 >$
11: \hspace{2em} $M = M . c_i < a_i >: T < \text{bif}_2 > \Rightarrow PR_i < \text{bif} >$
12: \hspace{2em} $T = T . e_i < \text{v}_i > \Rightarrow v_i . v_i \sim e_i < \in >$
13: \hspace{2em} $T = T . a_i < \text{out} > \Rightarrow e_i < \text{out} > \Rightarrow I < \text{bif}_1 >$
14: \hspace{2em} $T = T . a_i < \text{out} > \Rightarrow e_i < \text{out} > \Rightarrow I < \text{bif}_1 >$
15: \hspace{1em} \textbf{end if}
16: \textbf{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 $c_1 < a_1 >: e_1 < \text{out} > \Rightarrow e_T < \text{in} > . e_T < \text{out} > \sim v$ with $1 : e_1 & e_T \sim e'_T . 1 : e_1 & e_T \sim e'_T . e_1 < \text{out} > \sim v$

Transaction relations between $e_T$ and $e_2$ and data relations of $e_2$ as follows

$c_2 < a_2 >: e_T < \text{out} > \Rightarrow e_2 < \text{in} > . e_2 < \text{out} > \sim v_2$

are replaced by control and data relations $1 : e_2 & e'_T \sim e'_T . 1 : e_2 & e'_T \sim e'_T . e_2 < \text{out} > \sim v_2$

The notification transaction between $e'_T$ and $e''_T$ is also replaced similarly using the link resolution rule. Figure 12(b) shows 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 $e'_T$ and from $e_T$ to $e'_T$. As a result, $e_T$ does not have anything executing after it and 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 $e'_T$ using the identity elimination rule. In Step 4, we use the RCDE rule to remove the redundant control dependency from $e_1$ to $e''_T$. This is because the control dependency is cause by the relation $1 : e_1 & e'_T \sim e''_T$ and $e_1 > e'_T$, 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 refinement.

Fig. 12. Proof steps for transducer insertion during transaction routing refinement
resolution of a direct link from \( e_1 \) to \( e_2 \). Therefore, we have proved that the transducer insertion algorithm is functionality preserving.

<table>
<thead>
<tr>
<th>Application</th>
<th>High TLM</th>
<th>Low TLM</th>
<th># Trans.</th>
<th>Verification Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Voice Codec</td>
<td>B:188, D:138, Q:202; CD:428; DD:453</td>
<td>B:204, D:144, Q:221; CD:463; DD:470</td>
<td>Total:7113 (Flat:4726, IE:1600, LR:727, RCDE:18, Str:42)</td>
<td>3.7 sec</td>
</tr>
</tbody>
</table>

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. 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