Automatic Generation of Equivalent Architecture Model from Functional Specification

Samar Abdi
Center for Embedded Computer Systems
UC Irvine, CA 92697
sabdi@cecs.uci.edu

Daniel Gajski
Center for Embedded Computer Systems
UC Irvine, CA 92697
gajski@uci.edu

ABSTRACT

This paper presents an algorithm for automatic generation of an architecture model from a functional specification, and proves its correctness. The architecture model is generated by distributing the intended system functionality over various components in the platform architecture. We then define simple transformations that preserve the execution semantics of system level models. Finally, the model generation algorithm is proved correct using our transformations. As a result, we have an automated path from a functional model of the system to an architectural one and we need to debug and verify only the functional specification model, which is smaller and simpler than the architecture model. Our experimental results show significant savings in both the modeling and the validation effort.

Categories and Subject Descriptors

C.5.4 [**Computer System Implementation**]: VLSI Systems

General Terms

Algorithms, Design, Languages, Theory, Verification

Keywords

System level design, Formal verification, Model Refinement

1. INTRODUCTION

With rising complexity of design, modeling has been pushed to system level of abstraction. These models represent design decisions that must be evaluated for exploring the design space. The first critical design decision is to distribute the functionality in the specification onto components in the target architecture, thereby requiring an architecture model to reflect that decision. We also need to ensure functional equivalence of the specification and architecture models written in system level design languages (SLDL).

One approach is to manually write both models and verify them for equivalence by checking for similar properties using techniques like bounded model checking [4]. However, this would require manual effort in model rewriting as well as re-verifying every time the architecture is modified. Another approach, as presented in this paper, would be to verify the smaller and simpler functional specification first and then automatically refine it to an equivalent architecture model. To enable generation of an equivalent model, we require formalisms to represent the system level models and proof for correctness of the refinement step. Formalisms grouped under process algebra like CSP [6] and CCS [8] have been extensively researched and have well defined execution semantics. We build on top of these formalisms and use existing notions of hierarchy, behaviors and channels, available in SLDLs like SpecC [5] and SystemC 2.0 [1].

Correct by construction techniques have been widely applied at RT Level to prove the correctness of high level synthesis steps [9] [3]. A complete methodology for correct digital design has been proposed in [7], but they only consider synchronous models which are insufficient at system level. More recently, such techniques have been employed in synthesis of OS drivers for embedded devices [10].

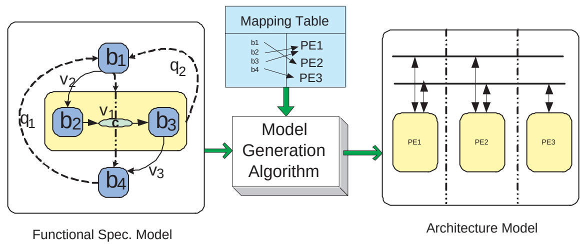

Figure 1: Architecture refinement in system design

Figure 1 shows how a specification model is refined to an architecture level model with communicating components in a parallel composition. Each task in the specification is mapped to a unique component in the architecture. The model generation algorithm, presented and proved in this paper, takes the functional specification and the mapping decision as its input. The output model carries behavior inside each of the components, that can be either compiled to assembly code (for a SW component) or synthesized to RTL (for a HW component). The rest of the paper is organized as follows. Section 2 will cover system level modeling constructs. Section 3 will present the architecture refinement algorithm. Section 4 will present notions of equivalence and functionality preserving model transformations followed by the proof of correctness for architecture refinement. Experimental results are shown in section 5. We will finally wind up with conclusions and future work.

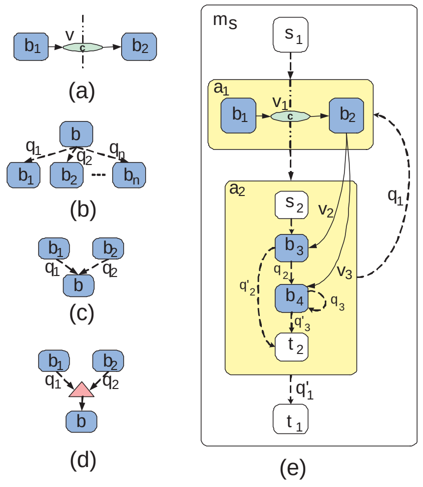

Figure 2: Modeling constructs and an example functional specification model m_s

2. SYSTEM LEVEL MODELING

Formally, a model is a set of objects and composition rules defined on the objects [2]. A system level model would have objects like behaviors for computation and channels for communication. The behaviors can be composed as per their ordering . The composition creates hierarchical behaviors that can be further composed. Interfaces between behaviors and channels or amongst behaviors themselves can be visualized as relations.

2.1 Modeling constructs

The various modeling constructs are shown in figures 2(a) through 2(d). Figure 2(a) shows data transaction between two parallel behaviors being realized through a channel. The channel implements two way blocking semantics, where both the sender and receiver wait until the transaction is completed. Data transfer between sequential behaviors is realized through ports. Figure 2(b) shows an ordered relation between behaviors. During execution, behavior $b_i, 1 \leq i \leq n$ may start executing after b has completed and condition q_i is TRUE. Similarly, in figure 2(c) behavior b may start executing if either b_1 is complete and q_1 is TRUE **or** b_2 is complete and q_2 is TRUE. In figure 2(d), the $\triangle$ construct blocks the execution of b until both b_1 and b_2 have completed and both q_1 and q_2 evaluate to TRUE.

Behaviors are either leaf-level(atomic) or hierarchical. Hierarchy is created either by parallel composition, like behavior a_1 in figure 2(e), or ordered composition like behavior a_2. The parallel execution semantics ensure that a_1 completes execution when both b_1 and b_2 are complete. The ordered composition indicates that a_2 starts with *start* behavior s_2 and completes when *terminate* behavior t_2 has completed. Child behaviors in ordered composition have arbitrary control amongst them like conditions and loops. Further, we introduce the notion of identity behaviors. An identity behavior is a leaf level behavior that does not perform any computation, and therefore its output is the same as its input. Such behaviors may be used for synchronization or retransmission of data. The *start* and *terminate* behaviors in ordered compositions are identity behaviors.

2.2 Notations

In order to express system models succinctly, we will use some simple notations. Hierarchical behaviors are expressed as functions of child behaviors. For instance, the parallel composition representing a_1 in figure 2(e) is written as $a_1 : \rho(b_1, b_2)$. The ordered composition for behavior a_2 is written as $a_2 : o(s_2, b_3, b_4, t_2)$. Control relations representing conditional execution of behaviors are written as $q.b_1 \rightsquigarrow b_2$, suggesting that b_2 may start after b_1 is complete and q is TRUE. If the execution order is unconditional, that is b_2 always executes after b_1, we write it simply as $b_1 \rightsquigarrow b_2$. Data transfer through ports is written as $v.b_1 \rightarrow b_2$, suggesting b_2 reads the data item v written by b_1. Transactions over channels are expressed with a pair of relations as $\{v.b_1 \rightarrow c, v.c \rightarrow b_2\}$, meaning that data v is sent from b_1 to b_2 over channel c. For the scope of this paper, we will use b and e for non-identity and identity behaviors respectively. Symbols q, v and c with suffix will be used for control conditions, data variables and channels respectively.

Using the above notation and the basic concepts of hierarchy, control and data flow, a system model m may be written as a three-tuple : $m :< H(m), C(m), D(m) >$, where $H(m)$ is the expression for the hierarchical composition of behaviors in m, $C(m)$ is the set of control relations in m and $D(m)$ is the set of data flow relations in m. The functional model m_s in figure 2(e) is thus written as follows.

$$\begin{aligned}
H(m_s) &= o(s_1, a_1 : \rho(b_1, b2), a_2 : o(s_2, b_3, b_4, t_2), t_1) \\
C(m_s) &= \{s_1 \rightsquigarrow a_1, a_1 \rightsquigarrow a_2, q_1.a_2 \rightsquigarrow a_1, \\
&\quad q_1'.a_2 \rightsquigarrow t_1, s_2 \rightsquigarrow b_3, q_2 : b_3 \rightsquigarrow b_4, \\
&\quad q_2'.b_3 \rightsquigarrow t_2, q_3.b_4 \rightsquigarrow b_4, q_3'.b_4 \rightsquigarrow t_2\} \\
D(m_s) &= \{v_1.b_1 \rightarrow c_1, v_1.c_1 \rightarrow b_2, v_2.b_2 \rightarrow b_3, v_3.b_2 \rightarrow b_4\}
\end{aligned}$$

3. DERIVING ARCHITECTURE MODEL

The goal is to generate a model that represents the mapping of system functionality to architecture components. The specification model is an arbitrary hierarchy of behaviors representing the system functionality. Model refinement would distribute the behaviors onto components that run concurrently in the system. It must be noted that refinement does not in any way influence the mapping decision. The designer is free to choose any mapping of behaviors to components and refinement would produce a model that represents it. However, each leaf behavior in the specification model must be mapped to only one component.

In order to derive an architecture model m_a from a specification model m_s, we use the designer decision of behavior mapping . The mapping can be written as a grouping of non-identity leaf behaviors in m_s. Let $\{b_1, b_2, ...b_n\}$ be the non-identity leaf behaviors of m_s. We can write $H(m_s) = f_s(b_1, b_2, ...b_n)$, where f_s is some function using the parallel and ordered composition rules. Let the system architecture consist of k components. Let $comp_i$ be the set of leaf behaviors mapped to i^{th} component. The construction of architectural model m_a for a given mapping is shown in the following algorithm.

An intuitive explanation of the refinement process is as fol-

Algorithm 1 Generate Architecture Model

```
1: $H(m_a) = \rho(pe_1, pe_2, ..., pe_k)$
2: $C(m_a) = \{\}$
3: $D(m_a) = D(m_s)$
4: for $i = 1$ TO $k$ do
5:   for $j = 1$ TO $n$ do
6:     if $b_j \in comp_i$ then
7:       $b_{ji} = o(s_{ji}, b_j, z_{ji}, t_{ji})$
8:       $z_{ji} = \rho(e_{j1}, e_{j2}, ..., e_{ji-1}, e_{ji+1}, ..., e_{jk})$
9:       $C(m_a) = C(m_a) \cup \{s_{ji} \rightsquigarrow b_j, b_j \rightsquigarrow z_{ji}, z_{ji} \rightsquigarrow t_{ji}\}$
10:    else
11:      $b_{ji} = \rho(e'_{ji})$
12:      $D(m_a) = D(m_a) \cup \{0.e_{ji} \rightarrow c_{ji}, 0.c_{ji} \rightarrow e'_{ji}\}$
13:    end if
14:  end for
15:  for all $q.x \rightsquigarrow y \in C(m_s)$ do
16:    $C(m_a) = C(m_a) \cup \{q.x_i \rightsquigarrow y_i\}$
17:  end for
18:  $pe_i = f(b_{1i}, b_{2i}, ..., b_{ni})$
19: end for
20: for all $v.b_i \rightarrow b_j \in D(m_s), b_i \in comp_l, b_j \in comp_r$ do
21:   if $l \neq r$ then
22:     $D(m_a) = D(m_a) - \{v.b_i \rightarrow b_j\}$
23:     $D(m_a) = D(m_a) \cup \{v.b_i \rightarrow e_{ir}, v.e_{ir} \rightarrow c_v, v.c_v \rightarrow e'_{ir}, v.e'_{ir} \rightarrow b_j\}$
24:   end if
25: end for
```

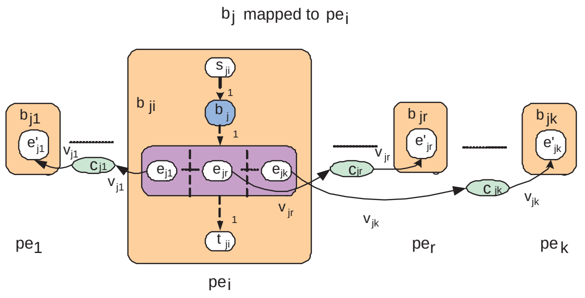

Figure 3: Architecture refinement applied to a single behavior

lows. We start by copying the behavior hierarchy of m_s onto behaviors pe_1 through pe_k, that represent the processing elements executing in parallel. Rename the sub-behaviors under each PE to reflect the mapping. For instance, b_1 copied under pe_2 is renamed as b_{12}. In order to realize the behavior mapping and to preserve the original control flow, we modify leaf behaviors under each PE as follows. If b_j is mapped to $comp_i$, then in the hierarchy under pe_i, replace b_j by a sequential composition $b_{ji} = o(s_{ji}, b_j, z_{ji}, t_{ji})$, where z_{ji} is a parallel composition of $k-1$ identity behaviors as shown in figure 3. Each of these identity behaviors is responsible for sending a message to other components that b_j has completed execution. For remaining components, replace b_j in the respective hierarchy by an identity behavior that receives the message for b_j's completion. Finally, all data transfers across components are routed through the identity behaviors used for synchronization. Intra-component data transfers are preserved as is.

The architecture model for the example in figure 2(e), as

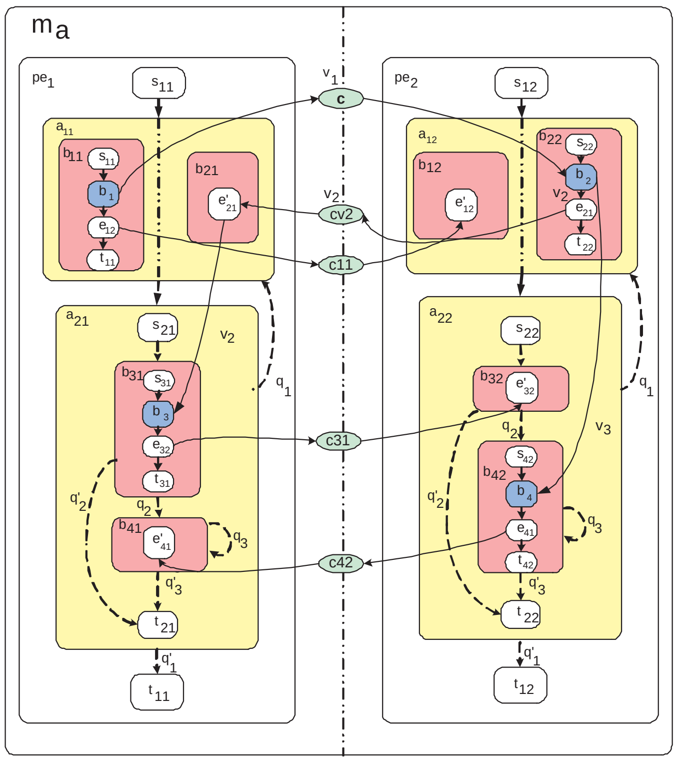

Figure 4: Architecture model m_a

derived by the refinement algorithm is shown in figure 4. In this particular model, the designer choose two components PE1 and PE2 for the system architecture. Behaviors b_1 and b_3 are mapped to PE1, while b_2 and b_4 are mapped to PE2.

4. PROOF OF CORRECTNESS

In order to prove correctness of the above model refinement algorithm, we must establish a notion of functional equivalence. This requires a definition of model execution semantics. Further, we present transformations that generate functionally equivalent models. The transformations are then used to present a proof of correctness.

4.1 Model execution semantics

The execution of a model is best understood by unfolding it as shown in figure 5(a). We construct a (possibly infinite) directed acyclic graph (DAG) representing all possible execution scenarios. Note that in this DAG, we have flattened the model. As a result of this flattening, the parallel composition (behavior a) is now modified to an equivalent partial order ($s_3 \rightsquigarrow b_1, s_3 \rightsquigarrow b_2$). A synchronization node is added to ensure that the execution of t_3 does not proceed until both b_1 and b_2 are complete. The label on the edges connecting behavior nodes are boolean variables or boolean constants, representing conditions for a behavior to execute. By default, the unlabeled edges represent a TRUE path. The index of the conditions represents the particular instance. A behavior node will execute if all its predecessors in the DAG have executed and all the incoming condition arc labels evaluate to TRUE. Input and output data associated with behaviors is also shown with incoming and outgoing variable arcs respectively.

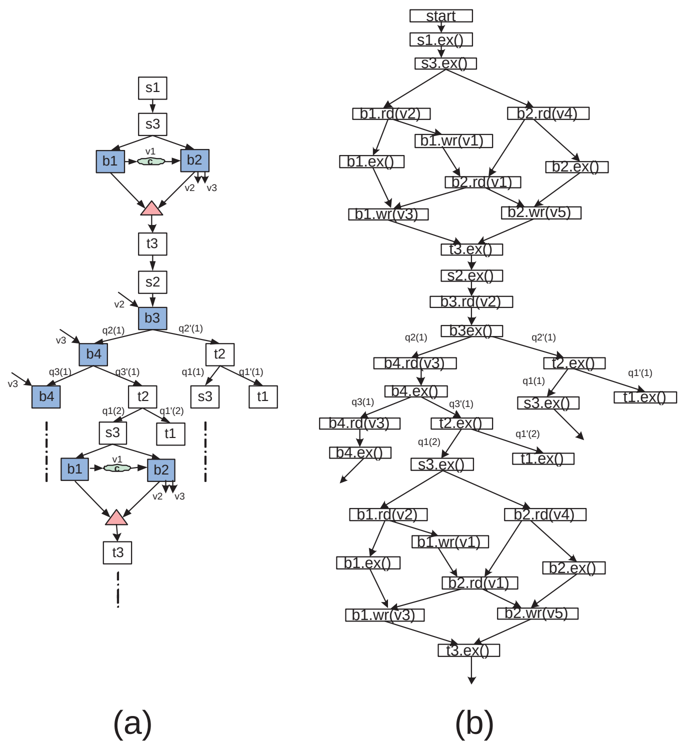
Figure 5: (a)Unfolded execution graph and (b) Action graph for specification model m_s

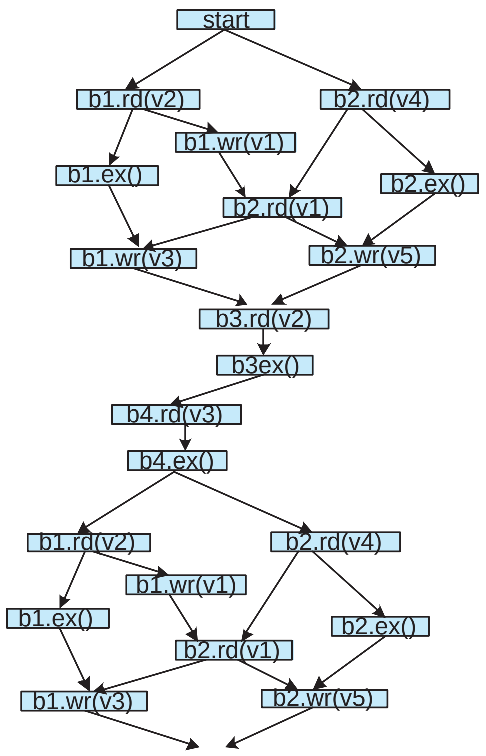
Figure 6: Partial order trace for an execution instance

4.1.1 *Action graph*

A behavior execution is further divided into three ordered sets of actions. First, the behavior reads all the input data (represented by b.rd(v)). Then the behavior executes its main body (represented by b.ex()). Any data transactions on the connected channels also take place interleaved with b.ex(). Finally, the behavior writes to all its output variables (represented by b.wr(v)). Note that the channel write and read actions are ordered as write followed by read, in compliance with the blocking channel semantics. Hence, we derive an *action graph* from the model execution graph as shown in figure 5(b) for our example.

4.1.2 *Partial order trace*

A model execution instance is simply a valuation of the conditional variables in the action graph. Given such a valuation, we can derive a partial order trace graph as shown in figure 6. In this particular execution instance, we have assumed a valuation to be $\{q_2(1) = T, q'_2(1) = F, q'_3(1) = T, q_3(1) = F, q_1(2) = T, q'_1(2) = F, ...\}$. Note that this trace contains only observable actions. Therefore all actions associated with identity behaviors are removed.

4.2 Equivalence of Models

We define functional equivalence of two models based on the above execution semantics. Two models are equivalent if they have the same
1. leaf level non-identity behaviors,
2. conditional variables, and
3. partial order trace for same valuation of conditions.
The equivalence of two models, say m_1 and m_2, by the above definition is written as $m_1 \leftrightarrow m_2$. Note that we define functional equivalence which is relevant for asynchronous system level models. Thus there is only a qualitative notion of time instead of a quantitative one. As models are refined towards greater detail, timing becomes more accurate and thus cannot be used quantitatively as a factor for equivalence.

4.3 Functionality preserving transformations

Considering the definition of functional equivalence, it can be seen that comparing two independent models for equivalence is intractable due to the potentially infinite size of the partial order traces. However, we can define some simple transformations on a model that produce functionally equivalent models. New models derived by applying a sequence of these transformations would thus be equivalent to the input model by induction. Figure 7 lists a set of model transformations that preserve functionality.

We now provide some intuition into the soundness of these transformations. Transformation T1 is sound by the semantics of parallel composition. A parallel composition can be turned into a partial ordered one by allowing all child behaviors to start together. Synchronization is then added to ensure that execution of the hierarchical behavior does not complete until all child behaviors are complete.

T2 replaces a control relation **to** a hierarchical ordered behavior with a control relation **to** its *start* behavior. Similarly, T3 replaces a control relation **from** a hierarchical ordered behavior with a control relation **from** its *terminate* behavior. By definition, the *start* and *terminate* behaviors are always the first and last, respectively, to be executed in an ordered composition. Thus both T2 and T3 are sound.

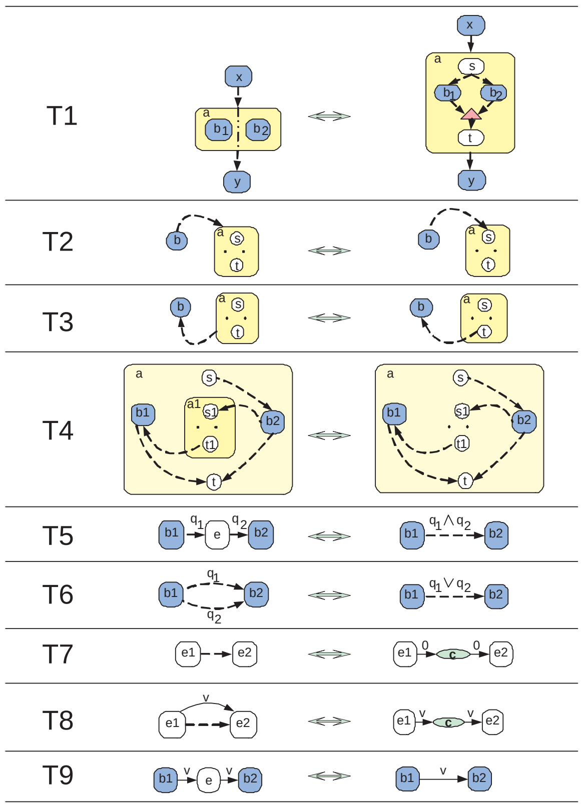

Figure 7: Model transformations

Transformation T4 flattens the hierarchy by removing a second level ordered composition that does not have any control relations. Since a hierarchical behavior itself is not observable in a partial order trace, it can be removed if it does not influence execution of other behaviors.

The LHS of T5 implies that if both conditions q_1 and q_2 are TRUE, then the behaviors $b_1, e,$ and b_2 are executed in that order. Since actions of identity behaviors are not included in the partial order trace, it is same as the trace for RHS, where y is executed after x if $q_1 \wedge q_2$ is TRUE. The LHS of T6 has two control relations, both leading from b_1 to b_2. If either of the condition variables q_1 or q_2 evaluate to true, then b_2 will be executed after b_1 completes. This is equivalent to a single control relation $(q_1 \vee q_2).x \rightsquigarrow y$ in the RHS.

According to blocking channel semantics, the RHS term in T7 would ensure that e_2 does not complete before e_1 starts. Since the actions of identity behaviors are not observed in the partial order trace, e_2 starting after e_1 completes is equivalent to e_2 being blocked until e_1 starts. The LHS in T8 implies that action $e_1.wr(v)$ is followed by $e_2.rd(v)$ due to the ordering of behaviors. For the RHS, the same order of data transactions is maintained due to channel semantics. Since the actions of identity behaviors are not included in the partial order trace, the data transfer through the port is equivalent to the same data transfer through a channel transaction in this case. The soundness of T9 follows from the same logic as above. Both the LHS and the RHS represent a partial order trace with action $b_1.wr(v)$ followed by $b_2.rd(v)$.

4.4 Correctness of architecture refinement

The proof for algorithm 1 is performed using the sound model transformations discussed in section 4.3. The refined model can be generated either through algorithm 1 or through a series of transformations. Typically, the derivation through transformations is longer, and thus less efficient, than the algorithmic step. Therefore, in the implementation of the refinement tool, we use the algorithm, more so because the intermediate models from individual transformations are not of interest. However, the transformations are essential for deriving the proof.

To prove equivalence, we reduce both models to a flat canonical form. The canonical representation of the architecture model is then simplified to optimize away redundant identity behaviors and channels. After optimization, the canonical form representation of the architecture model is reduced that of the specification model.We present here an intuitive version of the proof for lack of space.

The canonical form of a given model m is derived by converting all parallel compositions in the behavior hierarchy $H(m)$ to ordered compositions using T1. All control relations in $C(m)$ are then reduced to control relations only between leaf level behaviors by using T2 and T3. The hierarchy is then flattened by optimizing away the hierarchical sub-behaviors in $H(m)$ using T4. We thus get an equivalent model m' in the canonical form.

We start with models m_s and m_a and derive their canonical forms m'_s and m'_a respectively. So, we have $m'_s \leftrightarrow m_s$ and $m'_a \leftrightarrow m_a$. Now consider model m'_a. Given leaf level behaviors b_i, b_j in m'_s such that $q.b_i \rightsquigarrow b_j \in C(m'_s)$
Let $b_i \in comp_l, b_j \in comp_r, l \neq r$.
According to the refinement algorithm, we have
$\{b_i \rightsquigarrow z_{il}, q.e'_{ir} \rightsquigarrow b_j\} \subset C(m'_a)$, $0.e_{ir} \rightarrow e'_{ir} \in D(m'_a)$
Using T7 to replace the channel by the control condition, we get $\{b_i \rightsquigarrow z_{il}, e_{ir} \rightsquigarrow e'_{ir}, q.e'_{ir} \rightsquigarrow b_j\} \subset C(m'_a)$,
$D(m'_a) = D(m'_a) - \{0.e_{ir} \rightarrow e_{ir}\}$
Now, flattening z_{il} applying T5 twice gives us
$\{b_i \rightsquigarrow e_{ir}, e_{ir} \rightsquigarrow e'_{ir}, q.e'_{ir} \rightsquigarrow b_j\} \subset C(m'_a)$,
$= \{b_i \rightsquigarrow e_{ir}, q.e_{ir} \rightsquigarrow b_j\}$
$= q.b_i \rightsquigarrow b_j$

Now for the case when both b_i and b_j are mapped to the same component. Let $b_i, b_j \in comp_l$.
According to the refinement algorithm, we have
$\{b_{il} \rightsquigarrow b_{jl} \subset C(m'_a)$
Flattening by T4 gives us
$\{b_i \rightsquigarrow z_{il}, z_{il} \rightsquigarrow t_{il}.q.t_{il} \rightsquigarrow s_{jl}.s_{jl} \rightsquigarrow b_j \subset C(m'_a)$.
Finally, applying T5 and T6 gives us
$\{b_i \rightsquigarrow t_{il}, q.t_{il} \rightsquigarrow s_{jl}.s_{jl} \rightsquigarrow b_j \subset C(m'_a)$.
$= \{q.b_i \rightsquigarrow s_{jl}, s_{jl} \rightsquigarrow b_j\}$
$= q.b_i \rightsquigarrow b_j$

Using the above rules, we can reduce all conditional relations and synchronization channels in m'_a to those in m'_s. We now try to reduce the data flow relations across components.

Given leaf level behaviors b_i, b_j in m'_s such that $v.b_i \rightarrow b_j \in D(m'_s)$
Let $b_i \in comp_l, b_j \in comp_r, l \neq r$. In our refinement algorithm, data transfer across components was converted to data transactions over channels. Since b_i and b_j were mapped to different components, we have
$\{v.b_i \rightarrow e_{ir}, v.e_{ir} \rightarrow c_v, v.c_v \rightarrow e'_{ir}, v.e'_{ir} \rightarrow b_j\} \subset D(m'_a)$
Using T8, the transaction over channel can be reduced to simple port transfer with control condition, giving us

Table 1: Experimental results for different system architectures

Design	Model Configuration	Lines of Code	Modified LOC	Refinement time (auto)	Refinement time (manual)	Simulation time	Simulation Overhead
Jpeg	spec.	1932	-	-	-	0.322s	-
	arch.(2 PEs)	2841	1112	0.164s	22.24 days	0.445s	38.2%
	arch.(3 PEs)	4155	2306	0.259s	46.12 days	0.520s	61.5%
	arch.(4 PEs)	4342	2884	0.285s	57.68 days	0.640s	98.8%
Vocoder	spec.	9787	-	-	-	1.651s	-
	arch.(2 PEs)	12650	4618	0.746s	92.36 days	2.277s	37.9%
	arch.(3 PEs)	14874	7009	2.156s	140.12 days	2.351s	42.4%
	arch.(4 PEs)	18648	12665	10.679s	253.3 days	2.963s	79.5%

$\{v.b_i \rightarrow e_{ir}, v.e_{ir} \rightarrow c_v, v.c_v \rightarrow e'_{ir}, v.e'_{ir} \rightarrow b_j\}$
$= \{v.b_i \rightarrow e_{ir}, v.e_{ir} \rightarrow e'_{ir}, v.e'_{ir} \rightarrow b_j\}$
since $e_{ir} \rightsquigarrow e'_{ir} \in C(m'_a)$
Finally, applying T9 twice, we have
$\{v.b_i \rightarrow e_{ir}, v.e_{ir} \rightarrow e'_{ir}, v.e'_{ir} \rightarrow b_j\}$
$= \{v.b_i \rightarrow e'_{ir}, v.e'_{ir} \rightarrow b_j\}$
$= \{v.b_i \rightarrow b_j\}$

Since data transactions between behaviors mapped to the same component are preserved in the architecture model, we can reduce all data flow relations in m'_a to those in m'_s. We thus have $m'_a \leftrightarrow m'_s$. Using the equivalence result of the canonical form , we get $m_a \leftrightarrow m_s$.

5. EXPERIMENTAL RESULTS

In order to evaluate our claims about savings in model rewriting and validation effort, a model refinement tool was written in C++ based on the algorithm in section 3. Models were written in the SpecC language [5] and the tool was used to automatically create models for different architecture configurations. Tests were done with Jpeg encoder specification and a voice codec application based on the ETSI GSM Vocoder standard. Table 1 shows results for the tested configurations. The savings in model rewriting can be seen by comparing the effort in manual refinement versus automatic refinement. Using an optimistic metric of 50 correctly modified lines of code per person-day, manual refinement may take days or even months. In contrast, automatic refinement produces resulting model in seconds.

The models were simulated on a 2 GHz Linux machine using bitmap pictures for jpeg encoding and 3.7 second speech samples for the vocoder. Average simulation times per picture/voice sample are given in Table 1. Simulation overhead is calculated as the extra time for simulating architecture models over functional specifications. It can be seen that as the architecture becomes more complex, the simulation overhead increases. Moreover, the architecture model becomes difficult to debug due to several concurrent threads resulting from independently running components.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a method for generating an architecture level model from a functional specification model and proved its correctness. We established a notion of functional equivalence in the form of partial order traces and functionality preserving transformations were defined on system level models. Finally, the refinement algorithm was proven to be correct using these transformations. The generated architecture model can be used in several ways. The resulting data transaction channels at the top-level can provide estimates of communication traffic. This information can be used in building an optimal communication architecture. Also, the generated code for each of the components can be used as reference code either for software generation on processors or HDL code generation on HW components. In the future, we would like to enhance our scheme by extending the modeling constructs and the set of sound transformations to prove correctness of more design steps.

7. REFERENCES

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

[2] S. Abdi and D. Gajski. System Debugging and Verification: A New Challenge. Technical Report ICS-TR-03-31, University of California, Irvine, October 2003.

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

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

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

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

[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.

[10] S. Wang and S. Malik. Synthesizing operating system based device drivers in embedded systems. In *Proceedings of the International Symposium on System Synthesis*, pages 37–44, September 2003.