Provably Correct Architecture Refinement

Samar Abdi and Daniel Gajski

Technical Report CECS-03-29
September 30, 2003

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{sabdi,gajski}@cecs.uci.edu

Abstract

This paper presents a formal approach to correctly generate an architecture level model of a system from its specification model. We define the notion of equivalence of models based on their execution semantics. A formalism is then presented, which can be used to model systems and perform correct transformations on them. Architecture refinement is described, as a sequence of such transformations on the specification model, that results in an equivalent architecture model. This method of deriving one model from another through well defined rules can alleviate the problem of validating every model at different abstraction levels in system design.

Contents

List of Figures

Provably Correct Architecture Refinement

Samar Abdi and Daniel Gajski
Center for Embedded Computer Systems
University of California, Irvine

Abstract

This paper presents a formal approach to correctly generate an architecture level model of a system from its specification model. We define the notion of equivalence of models based on their execution semantics. A formalism is then presented, which can be used to model systems and perform correct transformations on them. Architecture refinement is described, as a sequence of such transformations on the specification model, that results in an equivalent architecture model. This method of deriving one model from another through well defined rules can alleviate the problem of validating every model at different abstraction levels in system design.

1. Introduction

The continuous increase in size and complexity of SoC designs has raised the abstraction level of system specification. We are thus faced with a new challenge, namely, how to make sure that abstract functional models written in languages like C++ can be synthesized to equivalent cycle-accurate implementations.

One approach is to use a system synthesis tool and use formal verification techniques to check for equivalence. In this approach, the designer specifies some properties that he or she expects to hold in the system. Techniques like model checking [2] and theorem proving [3] are then employed to check for these properties in the model. If a "golden" specification model of the system is available, techniques like FSM equivalence checking [10] may be used to compare the implementation model against the specification model. Although these techniques work well for abstraction levels of RTL and below, there are several problems in applying them to system level. Model checking requires a state transition system representation of the model, which may not always be available. Besides, it suffers from the state explosion problem. Theorem proving, though applicable at any abstraction level and on any kind of design, comes with minimal automation and requires extensive training and expertise. FSM equivalence checking also has problems in handling large designs and is not designed for asynchronous models that are used at system level. Moreover, checking functional equivalence of two independently written high level language programs is not feasible. **The only solution to the problem is deriving the refined model from the specification model through correct well defined refinement steps**.

A methodology based on well defined refinements [5] requires formalisms to represent the system level models and proofs for all refinement steps. Formalisms based on process algebra [4], like CSP [6], CCS [7] and ACP [1] have been around for quite some time and are suitable for representing system models. However, there were two problems we faced while evaluating these formalisms for our use. Firstly, we could not find a way to move processes across hierarchies, which was essential for our refinement steps. Secondly, our design methodology required clear separation of communication objects and computation objects. Formalisms such as π-calculus [8], although developed for communication between processes, do not have explicit communication objects like channels for synchronized data transfer. To solve the problem, we developed a formalism called *Model Algebra* that can be used for representing system models and their transformations.

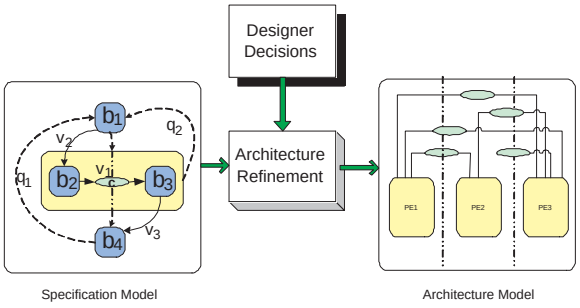

Figure 1. Architecture refinement in system design

Figure 1 shows how a specification model, with arbitrary hierarchical organization of tasks, is refined to an architecture level model with components communicating using abstract data transaction channels. The semantics of the archi-

tecture model ensures a parallel composition of components at the top level. The behavior inside each of these components can be either compiled to assembly code (for a SW component) or synthesized to RTL (for a HW component). Abstract data channels can be synthesized to system busses and bus interfaces on components through communication synthesis. The rest of the paper is organized as follows. Section 2 will cover system model semantics and the notion of model equivalence. Section 3 will introduce model algebra and its axiomatization. Section 4 will present the architecture refinement step and proof of its correctness in the context of model algebra. We will finally wind up with conclusions and future work.

2. System level models and equivalence

A system level methodology may be defined as a set of models with different semantics at different levels of abstraction. If the semantics are well defined, an abstract model may be refined to a more detailed one through a sequence of transformations. However, we must guarantee that this refinement produces a functionally equivalent model. We also need to define the notion of functional equivalence to establish the correctness of model transformations.

2.1. Specification model

Informal specification in natural languages are not executable and are often ambiguous. Therefore, they cannot be used as an input to a refinement tool. In our methodology, a specification is written as an executable model with well defined execution semantics. A model may be defined as a set of objects and their compositions. The two objects that we use are behaviors (representing computation) and channels (representing communication between behaviors). Behaviors may be composed hierarchically using composition rules of *ordered* and *parallel*. Behaviors can exchange data either through data variables or channels. Channels encapsulate data with synchronization to ensure sanity of data transaction between concurrent behaviors. Figure 2 shows a specification model. Ordered behaviors are connected by broken conditional arcs and can represent conditional execution, loops or any arbitrary FSM execution. They also have a unique start and terminate sub-behavior. Behaviors composed in parallel do not have any conditional arcs amongst them, and are shown as separated by dotted lines. Data transactions are shown as arcs labeled with data variables. Note that the specification model requires the behaviors to have a clean hierarchy, which allows us to treat the leaf level behaviors as indivisible units of computation.

Further, we introduce the notion of identity behaviors. An identity behavior is a leaf level behavior that does not perform any computation. In other words, an identity behavior has the same input and output. Such behaviors may be used for synchronization or retransmission of data. The *start* and *terminate* behaviors in ordered compositions are identity behaviors.

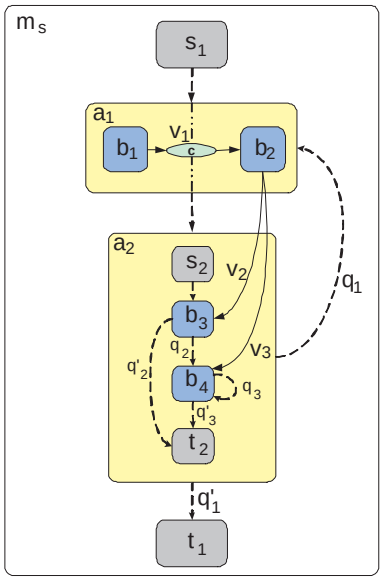

Figure 2. A Specification model

2.2. Execution semantics

The execution of a model is best understood by unfolding it as shown in figure 3. We construct a (possibly infinite) directed acyclic graph representing execution of the model. The square nodes represent leaf level behaviors (indivisible tasks in the model) and oval nodes represent channels. The label on the arcs connecting behavior nodes are boolean variables or boolean constants, representing conditions for a behavior to execute. 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. The subset of this DAG, consisting of behavior nodes and conditional arcs only, is topologically sorted and divided into time steps. Each time step consists of at least one behavior that may be executed in that time step. Behaviors executing in the same step exchange data through channels.

A behavior execution is further divided into three partially ordered sets of actions. First, the behaviors 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 concurrent to b.ex(). Finally, the behavior writes to all its output variables (represented by b.wr(v)). The actions in a typical

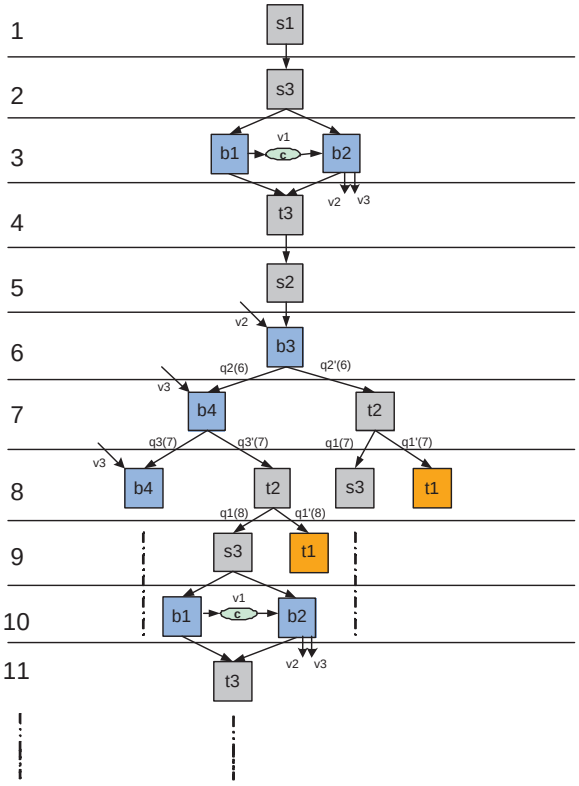

Figure 3. Unfolded execution graph for specification model

time step are shown in figure 4. Note that the channel write and read actions are ordered as write followed by read.

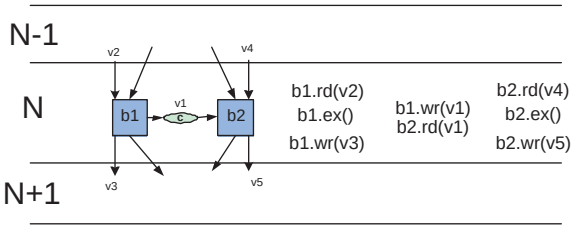

Figure 4. Partial order of actions in a time step

2.3. Model Equivalence

Using the execution semantics as explained above, we can derive a partial order of actions for any simulation run of the model. A simulation run of the model is nothing but a valuation of the (possibly infinite) set of conditional arc variables in the execution graph. Let Q_m be the valuation of the conditional arc variables and L_m be the set of non-identity leaf level behaviors in model m. We define **partial order trace** of m, with valuation Q_m as the partial order of actions of behaviors in L_m (written as $pot(m,Q_m)$).Two models m_1 and m_2 are **partial order trace equivalent** if

1. $L_{m_1} = L_{m_2}$
2. m_1 and m_2 have the same conditional arc variables
3. $\forall Q_{m_1} = Q_{m_2}, pot(m_1,Q_{m_1}) = pot(m_2,Q_{m_2})$

We will write $m_1 \leftrightarrow m_2$ if models m_1 and m_2 are partial order trace equivalent.

3. Model Algebra

We can now define our task as deriving an architecture level model m_a from a specification model m_s such that $m_a \leftrightarrow m_s$. It can be seen that because of the (possibly) infinite size of the execution graph, comparing two independently written models is not possible. We,therefore, define a formalism called *Model Algebra* for expressing models.

The signature of model algebra is as follows

$$MA = < B, C, \triangle, O, R >$$

B is the set of behaviors,
C is the set of channels,
$\triangle$ is the set of synchronization points,
$O = \{o, \rho\}$ (Set of Operations)
$R = \{\rightsquigarrow, \rightarrow\}$(Set of Relations)

We also define the class of identity behaviors as the subset B^{ε} of B. The intuitive explanation of the formalism is as follows. The operations of model algebra are defined on the set of behaviors and are used to create a behavioral hierarchy. The term $o(b_1,...b_n)$, where $b_1,...b_n \in B$, creates an ordered composition of behaviors. Similarly, the term $\rho(b_1,...b_n)$, where $b_1,...b_n \in B$ creates a parallel composition of behaviors. We also define the relation sub-behavior ($\triangleleft$) as follows. Let $a,b,t \in B$, then $b \triangleleft a$ if

1. $a = \rho(...b...)$
2. $a = o(...b...)$
3. $t \triangleleft a$ and $b \triangleleft t$

The relations of MA are used to create control flow and data flow between behaviors. The relation $q.b_1 \rightsquigarrow b_2$, where $b_1, b_2 \in B, q$ is a boolean variable, implies that behavior b_2 may start executing after b_1 completes and q evaluates to $TRUE$. The purpose of synchronization points ($\triangle$) is to force execution of a behavior only after all its predecessor behaviors (in the unfolded execution graph) have executed. For instance, the set of relations $\{q_1.b_1 \rightsquigarrow \delta, q_2.b_2 \rightsquigarrow \delta, \delta \rightsquigarrow b_3\}$, where $b_1, b_2, b_3 \in B, \delta \in \triangle$ implies that b_3 can start executing only after b_1 is complete and q_1 is $TRUE$ **AND** b_2 is complete and q_2 is $TRUE$.

Data flow between behaviors, either directly through variables or through channels, is represented using the $\rightarrow$ relation. The relation $v.b_1 \rightarrow b_2$, where v is a data variable implies the actions $b_1.wr(v)$ and $b_2.rd(v)$ in the respective order in execution trace. The pair of relations $\{v.b_1 \rightarrow c, v.c \rightarrow b_2\}$, where $c \in C$ implies data write and read through a channel.

3.1. Algebraic representation of models

For proving equivalence of models and formulas for refinement, algebraic representations are often times convenient. Using the notions of hierarchy, control flow and data flow of system models, we can represent a model as a 3-tuple in model algebra.

$$m :< H(m), C(m), D(m) >$$

$H(m)$ is the expression for 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 model algebraic representation of the specification model m_s in figure 2 is as follows.

$$\begin{aligned} H(m_s) &= o(s_1, a_1 : \rho(b_1, b2), \\ &\quad a_2 : o(s_2, b_3, b_4, t_2), t_1) \\ C(m_s) &= \{1.s_1 \rightsquigarrow a_1, 1.a_1 \rightsquigarrow a_2, q_1.a_2 \rightsquigarrow a_1, \\ &\quad q_1'.a_2 \rightsquigarrow t_1, 1.s_2 \rightsquigarrow b_2, q_2 : b_2 \rightsquigarrow b_3, \\ &\quad q_2'.b_2 \rightsquigarrow t_2, q_3.b_3 \rightsquigarrow b_3, q_3'.b_3 \rightsquigarrow t_2\} \\ D(m_s) &= \{v_1.b_1 \rightarrow c_1, v_1.c_1 \rightarrow b_2, v_2.b_2 \rightarrow b_3, \\ &\quad v_3.b_2 \rightarrow b_4\} \end{aligned}$$

3.2. Axioms of Model Algebra

We axiomatize model algebra modulo partial order trace equivalence. The axiomatization ξ_{MA} induces an equality relation on models that can be used to define correct transformations on models. For the following axioms, we assume $\{a, a_i, b, b_i\} \in B$, $\delta \in \triangle$, $\{s, t, e, e_i\} \in B^{\varepsilon}$, $c \in C$, $\{x, y\} \in B \times \triangle$, q_i is boolean variable and v_i is data variable. ϕ is a special data variable. If an object does not appear in $C(m)$ or $D(m)$, then it is said to have no relations. Control relations $1.x \rightsquigarrow y$ are written as $x \rightsquigarrow y$.

MA 1 $f(..a : \rho(b_1, ..., b_n)..), C(m), D(m) = f(..a : o(s, b_1, ..., b_n, t)..), C(m) \cup \bigcup_{i=1}^{n} \{s \rightsquigarrow b_i, b_i \rightsquigarrow \delta\} \cup \{delta \rightsquigarrow t\}, D(m)$

MA 2 $x \rightsquigarrow a : o(s, ...t) = x \rightsquigarrow s$

MA 3 $a : o(s, ...t) \rightsquigarrow x = t \rightsquigarrow x$

MA 4 $a_1 : o(s_1, .., a_2 : o(s_2, ...t_2), ..t_1) = a_1 : o(s_1, .., s_2, .., t_2, .., t_1)$ *iff* a_2 *has no relations.*

MA 5 $a_1 : \rho(s_1, .., a_2 : \rho(s_2, ...t_2), ..t_1) = a_1 : \rho(s_1, .., s_2, .., t_2, .., t_1)$

MA 6 $\{q_1.x \rightsquigarrow e, q_2.e \rightsquigarrow y\} = (q_1 \wedge q_2).x \rightsquigarrow y\}$

MA 7 $\{q_1.x \rightsquigarrow y, q_2.x \rightsquigarrow y\} = (q_1 \vee q_2).x \rightsquigarrow y\}$

MA 8 $e_1 \rightsquigarrow e_2 = \{\phi.e_1 \rightarrow c, \phi.c \rightarrow e_2\}$

MA 9 $\{e_1 \rightsquigarrow e_2, v.e_1 \rightarrow e_2\} = \{v.e_1 \rightarrow c, v.c \rightarrow e_2\}$

MA 10 $\{v.x \rightarrow e, v.e \rightarrow y\} = \{v.x \rightarrow y\}$

MA 11 $a : o(s, .., b_1, e, b_2, ..t) = a : o(s, .., b_1, b_2, ..t)$ *iff* e *has no relations.*

Soundness of ξ_{MA} means that the equality relation between models implies that they are partial order trace equivalent. Soundness is a must for the transformations to be correct. Completeness of ξ_{MA} implies that if any two models are equivalent, then their equality can be derived using the axioms of ξ_{MA}. Completeness is a highly desirable property because it ensures that all correct refinements can be proven to be correct. However, for the scope of this report, completeness is not necessary to be shown. The soundness of ξ_{MA} can be easily proved using the execution graph and partial ordered actions of models.

Theorem 1 ξ_{MA} *is sound modulo partial order trace equivalence.*

Proof

The first five axioms are essentially definitive. The execution graph is derived from the leaf level behaviors of the model. Since we restrict the model to clean hierarchies only, the composite behaviors are simply an encapsulation of the leaf behaviors. Models (and terms) on the LHS and RHS of axioms $MA1$ through $MA5$ define how the execution graph is constructed from the hierarchical model.

The LHS of axiom $MA6$ implies that if both conditions q_1 and q_2 are TRUE, then the behaviors $x, e,$ and y are executed in that order. Since the actions of identity behaviors are not included in the partial order trace, it will be the same as the trace for RHS, where y is executed after x if $q_1 \wedge q_2$ is TRUE.

The LHS of axiom $MA7$ has two control relations, both leading from x to y. If either of the condition variables q_1 or q_2 evaluate to true, then y will be executed after x. This is equivalent to a single control condition $(q_1 \vee q_2).x \rightsquigarrow y$ in the RHS.

The special data variable ϕ is used to replace a control relation with a data channel. According to channel semantics, the RHS term in $MA8$ would ensure that $e_2.ex()$ is followed by $e_1.wr(\phi)$ and $e_2.rd(\phi)$. Thus e_2 does not complete before e_1 starts. Since the actions of identity behaviors

are not included in the partial order trace, the trace for RHS will be the same as that for LHS.

The LHS in $MA9$ 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 actions is maintained. Since the identity actions are not included in the partial order trace, the trace for RHS will be the same as that for LHS.

The soundness of $MA10$ follows from the same concept as above. Both the LHS and the RHS represent a partial order trace with action $x.wr(v)$ followed by $y.rd(v)$.

In axiom $MA11$, we assume that e has no relations. Thus for the RHS, identity behavior e does not have any actions. Thus the partial order trace for RHS will be the same as that for LHS.

We have thus proven the soundness of ξ_{MA}.

4. Architecture Refinement

The architecture refinement task, as depicted in figure 1 is to generate a model that represents the mapping of system tasks to architectural components. The specification model is an arbitrary hierarchy of behaviors representing the system functionality. Architecture 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.

4.1. Deriving the architecture model

In order to derive an architecture model m_a from a specification model m_s, we use the semantics of the two models and the designer decision of behavior mapping . The designer decision can be written as a grouping of non-identity leaf behaviors in m_s. Let $\{b_1, b_2, \ldots b_n\}$ be the non-identity leaf behaviors in L_{m_s}. We can write $H(m_s) = f_s(b_1, b_2, \ldots b_n)$, where f_s is some function using the operations in model algebra. Let the system architecture consist of k components. Let $comp_i \subset L_{m_s}$ be the mapping to i^{th} component. We can derive the architecture model m_a as follows.

Copy the hierarchy in m_s onto behaviors pe_1 through pe_k. Rename sub-behaviors such that $\forall x \triangleleft H(m_s)$, there is a corresponding $x_i \triangleleft pe_i, 1 \leq i \leq k$. Modify behaviors in L_{m_s} as follows.

Let $e_{ji}, e'_{ji} \in B^{\varepsilon}, 1 \leq j \leq n, 1 \leq i \leq k$.

$$\begin{aligned} b_{ji} &= o(e_{ji}, b_j, e'_{ji}) \text{ if } b_j \in comp_i \\ &= \rho(e_{ji}, e'_{ji}) \text{ otherwise} \\ pe_i &= f_s(b_{1i}, b_{2i}, \ldots b_{ni}) \\ H(m_a) &= \rho(pe_1, pe_2, \ldots, pe_k) \end{aligned}$$

The control and data flow relations can be obtained as follows. Start with $C(m_a)$ and $D(m_a)$ as empty sets and perform the following steps.

1. $\forall q.x \rightsquigarrow y \in C(m_s), C(m_a) = \{\bigcup_{i=1}^{k} q.x_i \rightsquigarrow y_i\}$
2. $C(m_a) = C(m_a) \cup \{\bigcup_{j=1}^{n} \{e_{ji} \rightsquigarrow b_j, b_j \rightsquigarrow e'_{ji}\}\}$ where $b_j \in comp_i$
3. $\forall v.b_i \rightarrow b_j \in D(m_s)$, where $b_i \in comp_l, b_j \in comp_r$
   if $l = r, D(m_a) = D(m_a) \cup \{v.b_i \rightarrow b_j\}$
   if $l \neq r, D(m_a) = D(m_a) \cup$
   $\{v.b_i \rightarrow e_{jl}, v.e_{jl} \rightarrow c_v, v.c_v \rightarrow e_{jr}, v.e_{jr} \rightarrow b_j\}$
4. if $b_i \in comp_l, D(m_a) = D(m_a) \cup$
   $\{\bigcup_{i=1}^{n} \{\bigcup_{j=1, j\neq l}^{k} \{\phi.e_{ij} \rightarrow c_{ij}, \phi.c_{ij} \rightarrow e_{il},$
   $\phi.e'_{il} \rightarrow c_{il}, \phi.c_{il} \rightarrow e'_{ij}\}\}$

Intuitively, the architecture refinement process can be described as follows. In step 1, we copy over the control relations from the specification model to each component. In step 2, we introduce the control relations for the ordered place holder behaviors in each component. One such example is shown in figure 5, where the control relations for behavior b_{ij} must ensure execution of e_{ij}, b_i and e'_{ij} in that order. In step 3, we introduce channels to carry all data transfer across components. Finally, in step 4, we introduce synchronization channels to maintain the execution order to be the same as in the specification model.

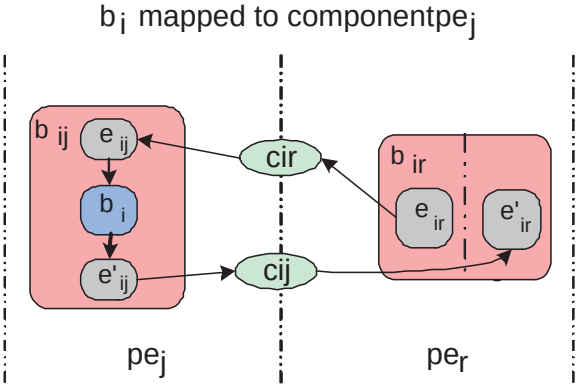

Figure 5. Architecture refinement applied to a single behavior

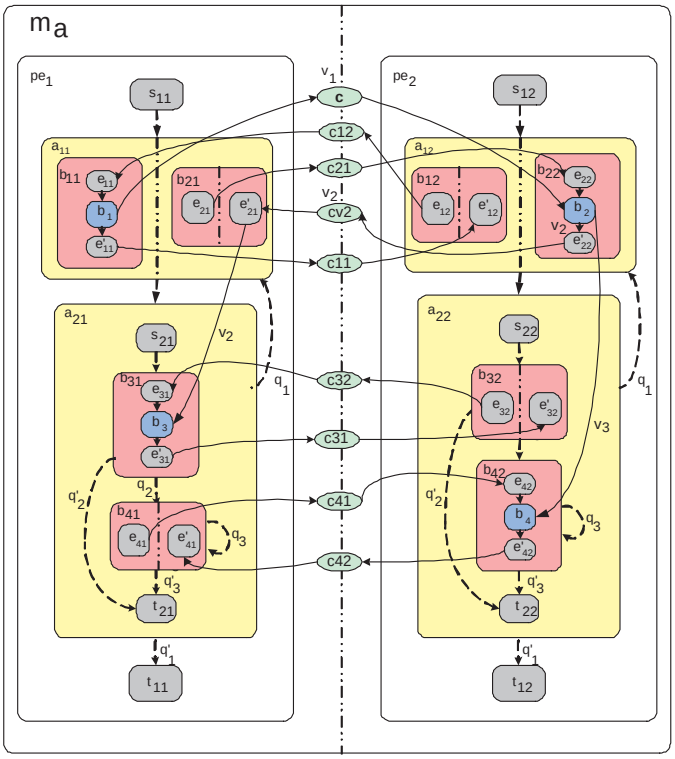

Figure 6. Architecture model m_a

The refinement function as applied to a single behavior is shown in figure 5. The complete architecture model for the specification model in figure 2 is shown in figure 6. 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.

4.2. Correctness of architecture refinement

The proof of correct refinement involves showing that the architecture model derived using the above steps is equivalent to the specification model. 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 algebraic representation of the architecture model is shown to be same as that of the specification model.

Theorem 2 $m_a = m_s$

Proof
We present here an intuitive and simplified version of the proof.

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 axiom MA 1. All control relations in $C(m)$ are then reduced to control relations only between leaf level behaviors by using axioms MA 2and MA 3. The hierarchy is then flattened by optimizing away the composite sub-behaviors of $H(m)$ using axioms MA 4 and MA 5. 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 $m'_s = m_s$ and $m'_a = m_a$. Now consider model m'_a.

Given $b_i, b_j \in L_{m'_s}, q.b_i \rightsquigarrow b_j \in C(m'_s)$
Let $b_i \in comp_l, b_j \in comp_r, l \neq r$.
By definition of the refinement steps in section 4.1, we have
$\{b_i \rightsquigarrow e'_{il}, q.e'_{il} \rightsquigarrow e_{jl}, e_{jr} \rightsquigarrow b_j\} \subset C(m'_a)$,
$\phi.e_{jl} \rightarrow e_{jr} \in D(m'_a)$
$\Rightarrow \{b_i \rightsquigarrow e'_{il}, q.e'_{il} \rightsquigarrow e_{jl}, e_{jl} \rightsquigarrow e_{jr}, e_{jr} \rightsquigarrow b_j\} \subset C(m'_a)$,
$D(m'_a) = D(m'_a) - \{\phi.e_{jl} \rightarrow e_{jr}\}$[MA 8]

However, $\{b_i \rightsquigarrow e'_{il}, q.e'_{il} \rightsquigarrow e_{jl}, e_{jl} \rightsquigarrow e_{jr}, e_{jr} \rightsquigarrow b_j\}$
$= q.b_i \rightsquigarrow e_{jl}, e_{jl} \rightsquigarrow b_j\}$ [MA 6]
$= q.b_i \rightsquigarrow b_j$ [MA 6]

Let $b_i, b_j \in comp_l$.
By definition of the refinement steps in section 4.1, we have
$\{b_i \rightsquigarrow e'_{il}, q.e'_{il} \rightsquigarrow e_{jl}, e_{jl} \rightsquigarrow b_j\} \subset C(m'_a)$.

However, $\{b_i \rightsquigarrow e'_{il}, q.e'_{il} \rightsquigarrow e_{jl}, e_{jl} \rightsquigarrow b_j\}$
$= \{q.b_i \rightsquigarrow e_{jl}, e_{jl} \rightsquigarrow b_j\}$ [MA 6]
$= q.b_i \rightsquigarrow b_j$ [MA 6]

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 $b_i, b_j \in L_{m'_s}, v.b_i \rightarrow b_j \in D(m'_s)$
Let $b_i \in comp_l, b_j \in comp_r, l \neq r$. By definition of the refinement steps in section 4.1, we have
$\{v.b_i \rightarrow e_{jl}, v.e_{jl} \rightarrow c_v, v.c_v \rightarrow e_{jr}, v.e_{jr} \rightarrow b_j\} \subset D(m'_a)$,
$e_{jl} \rightsquigarrow e_{jr} \in C(m_a)$
$\Rightarrow \{v.b_i \rightarrow e_{jl}, v.e_{jl} \rightarrow e_{jr}, v.e_{jr} \rightarrow b_j\} \subset D(m'_a)$
using [MA 9,MA 7]

However, $\{v.b_i \rightarrow e_{jl}, v.e_{jl} \rightarrow e_{jr}, v.e_{jr} \rightarrow b_j\}$
$= \{v.b_i \rightarrow e_{jr}, v.e_{jr} \rightarrow b_j\}$ [MA 10]
$= \{v.b_i \rightarrow b_j\}$ [MA 10]

Using the above rules, we can reduce all data flow relations in m'_a to those in m'_s. Since all the identity behaviors added during architecture refinement do not have any relations anymore, we can optimize them away by using transformation of axiom MA 11. We thus have $m'_a = m'_s$. Using the equality of canonical form , we get $m_a = m_s$.

5. Experimental Results

The theory of model algebra has shown us how to construct proofs for refinement algorithms. In order to demon-

Table 1. Experimental results for different system architectures

Design	Number of Components	Number of Leaf Behaviors	Number of Channels	Lines of Code	Refinement time	Simulation time
Jpeg	2	45	6	2841	0.164s	445s
	3	62	12	4155	0.259s	520s
	4	71	13	4342	0.285s	640s
Vocoder	2	117	8	12650	0.746s	2277s
	3	124	18	12874	2.156s	2351s
	4	142	59	18648	10.679s	2963s

strate the validity of these claims, an architecture refinement tool was written using the algorithm presented in 4. The unambiguous semantics of the models can be used to define refinement steps which can be automated.

The architecture refinement algorithm was implemented in C++ and experiments were done with specification models for Jpeg Encoder and GSM Vocoder. The JPEG specification had 2695 lines of code and consisted of 28 leaf behaviors. The Vocoder specification had 7992 lines of code and 62 leaf behaviors. Three architecture models were generated per design for architectures with 2, 3 and 4 processing elements. Table 1 shows the user times for refinement and simulations done with 1000 random test vectors for each model on a 2 GHz Pentium 4 machine. Note that as the number of components increase, the size and average simulation time of the architecture model increases.Even though these simulations are faster than those at cycle accurate level, exhaustive simulations for each intermediate model in the system design process would be a huge overkill.

6. Conclusions

We presented a method for generating an architecture level model from a specification model in a formal setting. We established a reasonable notion of equivalence in the form of partial order traces. A formalism was developed with the goal of creating correct transformations on models. Finally, architecture refinement was shown to be correct using the axioms of our formalism.

This approach, based on correct model transformations, shows a lot of promise in generation of equivalent system level models. It does not suffer from the memory explosion problem of state based approaches and also enables designers to define unambiguous semantics of models in their methodology. Developing the refinement steps and its proof is a one time effort compared to functional verification of every model in every design. This effort has already been used in developing tools for automatic generation of refined models in our system design methodology [9]. Future work in this direction would involve extending the theory to include data types and develop proofs for communication and cycle accurate model refinements.

References

[1] J. Bergstra and J. Klop. Process algebra for synchronous communication. In *Information and Control*, pages 109–137, 1984.

[2] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[3] D. Cyrluk, S. Rajan, N. Shankar, and M. Srivas. Effective theorem proving for hardware verification. In *Theorem Provers in Circuit Design*, pages 203–222, September 1994.

[4] W. Fokkinik. *Introduction to Process Algebra*. Springer, January 2000.

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

[8] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[9] J. Peng, S. Abdi, and D. Gajski. Automatic model refinement for fast architecture exploration. In *Proceedings of the Asia-Pacific Design Automation Conference*, pages 332–337, January 2002.

[10] T. M. S. Devadas and R. Newton. On the verification of sequential machines at different levels of abstraction. In *Proceedings of the Design Automation Conference*, pages 271–276, June 1987.