Decomposition of Sequential Behavior Using Interface Specification and Complementation

KAMLESH RATH, VENKATESH CHOPPELLA, STEVEN D. JOHNSON
Computer Science Department, Indiana University, Bloomington, Indiana, 47405

Decomposition of system behavior along functional boundaries into interacting sequential components is a key step in top-down system design. In this paper, we present sequential decomposition, a method for factoring sequential components from a system specification based on interface specifications of the components. The resulting components can be independently synthesized, or realized using off-the-shelf components. We introduce interface specification language (ISL), based on finite-state machine semantics, to specify the input/output behavior of synchronous sub-systems. A component is factored from a system by embedding an implementation of the complement of its interface into the system description. The composition of a machine with its complement is shown to be isomorphic to the machine, and the composition of a machine with an implementation of its component is shown to be a safe interaction. We apply sequential decomposition to a non-trivial example, a special-purpose computer with Scheme programming language primitives as its instructions.

Key Words: Interface protocol specification, Decomposition, Formal methods

1. INTRODUCTION

Decomposition of system specifications for computer-aided system design is an active topic in synthesis research. As Gajski points out, system synthesis by design partitioning and interface synthesis must be explored to make synthesis viable for large designs [1]. Synthesis of systems with complex control structures into designs with “monolithic” control units can result in unwieldy implementations. A design tool should have the flexibility to let a designer decompose a system into suitable components with explicit synchronization and value communication. Synthesis of these decomposed components with register transfer level specifications can then be accomplished by high-level synthesis techniques.

Derivation is a formalization of synthesis with more emphasis on “correct construction” than on design automation. Our tools constitute a set of transformations that are used to engineer an implementation from a specification, with each transformation accumulating information about the implementation. In a functional framework, a transformation called system factorization [2] was used earlier to extract functional components having naive interactions with the surrounding system. As a generalization of system factorization, we have developed sequential decomposition which uses a finite state machine model to decompose system descriptions into interacting sequential machines [3, 4].

We introduce Interface specification language (ISL), to describe the interaction of a machine with its environment, orthogonal to its functional behavior. An extensive presentation of the language is given in [3]. The complement of a machine specifies the behavior of its environment. We define implementation relations over machines. Decomposition is accomplished by encapsulating parts of a system at the algorithm, process, or operation level of granularity into an abstract component with a specified interface. An implementation of its complement is then embedded as the interface “stub” in the original system description.

A composition operation is defined on the machines to model the synchronous interaction between machines. It is shown that a machine composed with its complement results in a closed machine that is isomorphic to the original. It is also shown that a machine composed with an implementation of its complement results in a safe interaction, meaning that the machines can complete an interaction protocol and reach their respective final states.

1.1 Related Research

Several researchers have looked at the interface issues involved in system synthesis. Borriello uses timing
diagrams to specify the interface of a circuit and synthesis tools to generate the interface automatically [5]. While Boriello develops these external interface specifications as a means to guide synthesis, our goal is to use them to guide design decomposition. Jajnik and Ciesielski [6] perform top-down machine decomposition by partitioning outputs and states in state graphs with the objective of performance and area optimization of synthesized PLA circuits. Specification at different levels of abstraction and partitioning of control and data flow graphs for synthesis have been considered by Kuehlmann and Bergamaschi [7] to obtain smaller layouts. Our approach is to enable designers to decompose systems into logically and functionally distinct components and not to use heuristics to partition a design based on layout constraints.

Józwiak et al. [8, 9] have developed heuristic methods for simultaneous decompositions of sequential machines into component machines by partitioning the state space, inputs and outputs. The inter-connections between the components are also determined heuristically by analyzing the machine structure. This method can be used to decompose a machine into sub-machines based on various area and speed constraints, but it does not take into account timing and protocol constraints in a design.

System-level decomposition in the System Architect's Workbench is accomplished by behavioral transformations [10]. Walker and Thomas show transformations on the controller and selector to partition a design into processes. The processes created using their method have a very simple interaction scheme to transfer data values and control signals using message passing. Their approach cannot synthesize components using complex protocols for data transfers and synchronization.

SpecPart [11] partitions algorithm/process grained computations from the SpecChart behavioral specifications. Default protocols are used for interaction between components. The CHOP system-level design partitioner [12] uses task graphs to specify the protocol between every partition. Special purpose hardware units called data-transfer modules are used on both sides of each interaction. Although this method allows for complex protocols and use of off-the-shelf components, the interface has to be designed manually and may be expensive in terms of area and performance because of the special purpose modules.

In our approach, parts of a system can be abstracted and the protocol between the components can be incorporated into the components without using any special-purpose modules and without restrictions on the protocols. Components of a system can be independently synthesized, or mapped to off-the-shelf components such as dynamic RAMs and floating-point units by specifying their interface from timing diagrams.

In related formal methods research, Kurshan [13] verifies reactive systems by stepwise reduction and refinement using L-automata with language and process homomorphism. Gopalakrishnan et al. [14] have used an annotated state machine model for bottom-up hardware specification and synthesis in HOP. Drusinsky and Harel have used state-charts for bottom-up hierarchical specification [15] by embedding simpler state machines at a lower level of specification into states at a higher level of specification. The tree of state machines is then synthesized into a network of PLAs. Levin [16] uses a hierarchical automata model for system specification targeted towards a network of PLAs with memory. This method only supports naive interactions between constituent automata where one of the automata is in “operation” and all the others are “waiting for its response”.

Clarke et al. [17, 18, 19] have used a compositional finite state machine model to verify temporal properties of systems using computational tree logic and binary decision diagram based methods. Davie and Milne [20, 21] have used constraints on the target architecture and the context of a design in CIRCAL to reduce the complexity of verification. Dill et al. [22] have looked at asynchronous hardware verification using the Murph HDL and verifier to test for deadlocks and invariants. As an alternative to bottom-up verification techniques, our approach facilitates top-down design by factoring sequential components from designs using transformations.

1.2 Outline

The rest of this paper develops a theoretical basis for decomposition of sequential components from system specifications. Section 2 gives a brief introduction to Interface Specification Language and a finite state machine based semantic model for it. Section 3 describes the environment in which a machine operates, its complement machine. Section 4 gives a construction for composition of machines. The composition operation formalizes the synchronous interaction of machines for given port connections. We define a machine isomorphism property and prove that a machine composed with its complement results in a closed machine isomorphic to the original. Section 5 defines the implementation and path implementation relations by extending the inclusion relation over port values in machine transitions to states and machines. We also show that a machine composed with a path implementation of its complement results in a safe interaction, i.e. the final states of the constituent machines are reachable.

As an example, a scheme machine is used to illustrate the ideas in the paper. The Scheme machine is a special-purpose computer providing the symbolic pro-
cessing primitives of the Scheme programming language. The system includes a processor and a heap with a stop-and-copy garbage collector, an allocator, and a dynamic RAM based memory subsystem. We will use sequential decomposition on a high-level system description to derive the interactions between the CPU, allocator, and garbage collector in the system organization shown in Figure 1.

2. INTERFACE SPECIFICATION LANGUAGE

A definition in Interface Specification Language (ISL) is used to specify the input/output interaction of a machine with its environment. Communication between machines is modeled as values over connected ports. Other forms of communication, such as using buffers or shared storage, must be modeled explicitly. The interface is defined over input and output control ports (CI, CO) and input and output data ports (DI, DO). The ports can range over a set of values (V) which includes the don’t care value (#).

ISL is built on three kinds of synchronization primitives, lock-step, 1-way and 2-way synchronization. The operators sequence (;), choice (\[\]), interleave (\[\)) and repetition (*) are used to form expressions in the language. The complement (\[\)) and composition (M IM2)\[\]) operations are discussed in Sections 3 and 4. A detailed discussion of ISL syntax is presented in [3].

The BNF-style syntax description of the language is given in Figure 2. A machine is parametrized by the port names occurring in its defining expression. An action consists of a set of values on data ports guarded by certain truth values on control ports. Input actions \[\] denote a set of control inputs guarding data actions. Similarly, output actions \[\] denote a set of control outputs guarding data actions. The internal behavior of a machine can be annotated in any expression in the interface description.

2.1 Machine Model

The semantics of ISL is based on a finite-state machine model. A machine is a sextuple \[M = \{S, T, r, f, P, R, V\}\], where \(S\) is the set of states, \(T\) is a non-empty set of transitions, \(r\) is the reset/start state, \(f\) is the final state, \(P\) is the set of ports, \(R\) is a set of internal registers and \(V\) is the domain of values. The set of ports is a union of the sets of control inputs, control outputs, data inputs, and data outputs \([P = CI \cup CO \cup DI \cup DO]\). A machine with no input ports is called a closed machine.

The final state of a machine indicates the completion of protocols associated with the machine; it does not indicate termination. There are two kinds of states, transit states and wait states. A transit state has no transitions from the state to itself. A wait state has at least one transition from the state to itself. In state diagrams, wait states are indicated by \(\cdot\) and transit states are identified as \(\cdot\).

Transitions are indicated as: \(s_1 \subseteq s_2\), where \(s_1\) is the source state, \(s_2\) is the target state, and \(\subseteq\) is a label. A label \(\subseteq\) is an assignment function \(P \rightarrow R \rightarrow V\).

Definition 2.1: The care set for a label \(\subseteq\) is defined as: \(\text{care } \subseteq = \{ p \mid p \in P, \not\subseteq (p) \neq #\}\). It denotes the set of ports not assigned don’t care values by the label \(\subseteq\).

The semantics of a definition in ISL is the finite state machine it constructs. A detailed discussion of ISL is presented in [3].

2.2 Example—Garbage Collector

In the system shown in Figure 1, consider the interface of the garbage collector. It follows a simple protocol where it waits for collect to be asserted and then after a finite interval asserts gc-active.

\[
gc(\text{collect},\text{gc-active})(M)\{\}
\]

\[
[; \text{await collect}T; \text{compute gc-active}\{F,M=\text{garbage-collect}(M)\}]^{*}
\]

FIGURE 1 Scheme Machine System Organization.
Interaction $A ::= C : D$

- $\text{compute } A_o$
- $\text{await } A_i$
- $A_o \text{ until } A_i$
- $A_i \text{ before } A_o$

where $C ::= c_1/v_1, \ldots, c_n/v_n$

$D ::= d_1/v_1, \ldots, d_n/v_n$

$c_j \in CI \cup CO$, $d_k \in DI \cup DO$

$v_j, v_k \in V$

Expression $E ::= A \mid E \cdot A \mid E \parallel E \mid E \times E$

Machine $M ::= E \mid E^{\ast} \mid M \mid (M \parallel M)^N$ where $N ::= \text{port connectivity netlist}$

The ISL specification for the garbage collector is given above. Figure 3 shows the state machine for the garbage collector interface.

### 3. COMPLEMENTATION

In a system of interacting sequential components, no single component should be considered in isolation. A component must be specified in relation to its interaction with its surroundings. This interaction is typically specified by control and data signal interfaces. When isolating the single component, it is useful to view this isolation as an interaction between the component and a complement machine that abstracts the behavior of the environment. Notice then that the complement of a machine should be identical to itself, except with its input and output ports reversed and the internal registers hidden. This might suggest that the notion of complementation is a trivial one, since the interaction of a machine and its complement is already 'fixed'. However, by generalizing the connectivity of input and output ports, or by considering an implementation of the complement machine, the resultant composed machine may exhibit other interesting kinds of behavior.

We define a function "gensym" that generates a new input(output) port name for an output(input) port name.

**Definition 3.2:** Given a machine $M = (S, T, r, f, P, R, V)$ where $P = CI \cup CO \cup DI \cup DO$, the complement machine $\bar{M} = (S, T, r, f, \bar{P}, \phi, V)$ where $\bar{P}$ is a set of new port names, one for each port in $P$. The set of transitions

$\bar{T} = \{ s_1 \rightarrow s_2 | s_1 \rightarrow s_2 \in T \text{ and } \bar{T} = \text{Rename}(\bar{L}) \}$

where $\bar{L}(p') = \text{Rename}(L)p' = L(p)$ where $p' \in \bar{P}$ is the new port corresponding to $p \in P$. The "Rename" function creates an assignment function $\bar{L}$ corresponding to $L$ such that the corresponding complementary ports have the same value.

The complement machine is constructed by creating a complementary set of ports with the same value on corresponding ports. It retains the sets of values, states, reset and final states, but its internal behavior is unspecified. The complement of the garbage collector is shown in Figure 3.

![Garbage collector (a) and its complement (b).](https://via.placeholder.com/150)
4. COMPOSITION

The composition operation creates a restricted product machine for a given set of port connections. The composition operation on machines with respect to a particular connection of their ports is used to construct a machine that behaves as the constituent machines executing synchronously. The construction of the composed machine is similar to "lock-step cartesian product" in HOP[14]. Composition only creates reachable states and transitions starting inductively from the start state. In each inductive step, the transitions in each machine that can "occur synchronously" are used to form a transition in the composed machine. The states that the machines reach after taking these transitions make up a state in the composed machine. This state is then used to compose transitions in the next inductive step of the construction.

Given machines $M_1 = (S_1, T_1, r_1, f_1, P_1, R_1, V_1)$ and $M_2 = (S_2, T_2, r_2, f_2, P_2, R_2, V_2)$ with ports $P_1 \rightarrow C_{11} \rightarrow D_{11} \rightarrow D_{12}$ and $P_2 \rightarrow C_{21} \rightarrow D_{21} \rightarrow D_{22}$, we define the equivalence relation $\sim$ between $P_1 \rightarrow P_2$ as follows:

**Definition 4.3:** $p_1 \sim p_2$ iff $p_1, p_2$ are either both control or both data ports and $p_1, p_2$ are connected.

Each equivalence class $N$ of $\sim$ is called a net. Connecting input ports together creates an input net. All other combinations of port connections create output nets.

**Definition 4.4:** Given machines $M_1 = (S_1, T_1, r_1, f_1, P_1, R_1, V)$ and $M_2 = (S_2, T_2, r_2, f_2, P_2, R_2, V_2)$ and the net-list $\mathcal{N}$ between $P_1$ and $P_2$, the composed machine $(M_1 || M_2)_\mathcal{N}$ is constructively defined by $(S, T, r, f, P, V)$ where $P = C_{11} \cup C_{12} \cup D_{11} \cup D_{12}$.

Each equivalence class $N$ of $\sim$ forms a port in the composed machine. It is represented as $[p]$ where $p \in N$.

$CI = \{ N \mid N \text{ contains no control output ports} \}$
$CO = \{ N \mid N \text{ contains at least one control output port} \}$
$DI = \{ N \mid N \text{ contains no data output ports} \}$
$DO = \{ N \mid N \text{ contains at least one data output port} \}$

Let $\mathcal{L}_{12}(p) = \mathcal{L}_1(p)$ if $p \in P_1$

The set of states $S$ and transitions $T$ are constructed by the following inductive schema:

1. The start state of the composed machine $r$ is $\langle r_1, r_2 \rangle$.
2. The transition $\langle q, s \rangle \xrightarrow{a} \langle q', s' \rangle \in T$ and $\langle q', s' \rangle \in S$ is reachable iff:
   i. $q \xrightarrow{a} q' \in T_1$ and $s \xrightarrow{a} s' \in T_2$
   ii. $\langle q, s \rangle \in S$ is reachable
   iii. $\mathcal{L}_1, \mathcal{L}_2$ are composable, that is, there exists $j_1, j_2, \ldots, j_k$ such that
      a) $\text{care}_{j_1} \cup \text{care}_{j_2} \subseteq \bigcup_{i=1}^{k} \{p_i\}$

$\forall p, p' \in \{p_i\} \cdot \mathcal{L}_{12}(p) = \mathcal{L}_{12}(p')$

where the transition label for the composed machine is:

If the final states of two interacting machines are reachable, then we say that the composition of the two machines is safe. Both the machines must follow a correct protocol to reach their final states. The safety of the composed machine implies that there is some interaction sequence between its constituent machines that leads to the final state of the composed machine. It does not imply that all possible interactions between the machines will reach the final state.

**Definition 4.5:** The composed machine $(M_1 || M_2)_\mathcal{N}$ is considered safe if the state $\langle f_1, f_2 \rangle$ is reachable, unsafe otherwise.

4.1 Machine Isomorphism

**Definition 4.6:** Two machines $M_1$ and $M_2$ are isomorphic if $M_1$ and $M_2$ have isomorphic sets of states and transitions.

We will prove that the composition of a machine $M$ with its complement in which each port connected to its corresponding renamed port, results in a closed machine isomorphic to $M$.

**Theorem 4.1** Given a minimal deterministic machine $M = (S, T, r, f, P, R, V)$ where $P = C_{1i} \cup C_{12} \cup D_{11} \cup D_{12}$ and its complement machine $\overline{M} = (S, \bar{T}, r', f, \bar{P}, \bar{R}, \bar{V})$ and $\mathcal{N}_0$ is the port map obtained by connecting every port $p \in P$ with its complementary port $p' \in \bar{P}$. Let $M_0 = (M || \overline{M})_{\mathcal{N}_0}$. Then $M_0$ is a closed machine isomorphic to $M$.

**Proof:** The proof follows by induction on the length of the derivation of states and transitions in the construction of $M_0$. The proof is given in the Appendix.

We have shown that a machine composed with its complement results in a closed machine isomorphic to itself. There is a one-to-one mapping between the states and transitions in $M$ and $M_0$. There is not state space explosion in the composed machine. The observable temporal behavior of the composed machine $M_0$ is the same as $M$. The composed machine is closed and self-contained. Figure 4 shows the construction of the machine composed using the garbage collector and its complement with every port connected with its complementary port. We can see that the composed machine $(gc || gc)_{\mathcal{N}_0}$ is isomorphic to the original machine, and has no input ports.
5. IMPLEMENTATION

We would now like to explore relations over machines based on their input/output behavior. The sequence of transitions of a machine and the values associated with the ports in the machine for each transition are key to the implementation relations described in this section. Relations over the reset/start states of the machines and the intermediate states along the sequence of transitions are used to describe a strong implementation (⊕) relation and a weak path implementation (⊕p) relation.

A one-to-one mapping of the ports in \( M_1 \) and \( M_2 \) is the basis for the inclusion relation over transition labels. Let \( p_1 \) and \( p_2 \) be ports in \( M_1 \) and \( M_2 \) respectively. A map \( p_1 \mapsto p_2 \) means \( p_1 \) corresponds to \( p_2 \) and the values on these ports can be compared.

**Definition 5.7:** The inclusion relation over transition labels with respect to a port map is defined as:

\[
\mathcal{L}_1 \leq \mathcal{L}_2 \iff \text{for all } p_1 \in \mathcal{L}_1, p_2 \in \mathcal{L}_2, \text{ } p_1 \mapsto p_2 \text{ and } \mathcal{L}_1(p_1) = \mathcal{L}_2(p_2)
\]

The binary relation "simulated by" (⊕) is a maximal relation over states, \( S \subseteq S_1 \times S_2 \), where \( S_1, S_2 \) are the sets of states in \( M_1, M_2 \).

**Definition 5.8:** A relation over states is a simulation relation if \( s_1 \rightarrow \text{relation} \rightarrow s_2 \) implies:

\[
\text{for all } s_1 \xrightarrow{\mathcal{L}_1} s_1', \text{ there exists } s_2 \xrightarrow{\mathcal{L}_2} s_2' \text{ such that } \mathcal{L}_1 \subseteq \mathcal{L}_2 \text{ and } s_1' \subseteq s_2'
\]

A machine \( M_1 \) is implemented by (⊕) machine \( M_2 \) if every state in \( M_1 \) is simulated by some state in \( M_2 \), and the start state of \( M_1 \) is simulated by the start state of \( M_2 \). Let \( r_1, r_2 \) be the start states of \( M_1, M_2 \).

**Definition 5.9:** The relation \( M_1 \) implemented by \( M_2 \) (\( M_1 \subseteq M_2 \)) holds iff:

\[
r_1 \subseteq r_2 \text{ and for all } s_1 \in S_1 \text{ there exists an } s_2 \in S_2 \text{ such that } s_1 \subseteq s_2.
\]

5.1 Path Implementation

Each path from the start state to the final state in the complement machine represents a valid sequence of interactions to complete a protocol. A machine can interact with an implementation of any interaction path of its complement machine. Decomposition of a component from a system is accomplished by incorporating the appropriate path implementation of the complement of the component into the description of the rest of the system.

A path implementation of a machine implements one of many protocols of the machine. It is a weaker relation than implementation which implements all protocols in a machine. Consider for example an arithmetic unit which employs different protocols for different operations (e.g. plus and divide). Such a device would have distinct complements for distinct protocols—we call these partial complements path complements because they characterize different paths through the component's machine.

To define the path implementation relation we must first define the relation path simulates over states. The binary relation path simulates is a maximal relation over states, \( S_1 \subseteq S_1 \times S_2 \), where \( S_1, S_2 \) are the sets of states in the two machines. \( s_1 \subseteq s_2 \) implies four conditions (Definition 5.10). The intuition behind the first condition is that there are some transitions from both \( s_1 \) and \( s_2 \) that can interact and lead towards completion of the protocol. The second condition states that all transitions from \( s_1 \) and \( s_2 \) that can interact lead to states in the path simulation relation. This assures us that the machines will always reach states in the path simulation relation. The third condition states that for all transitions with active control inputs from \( s_1 \), there is a corresponding transition from \( s_2 \), with which it can interact and these transitions lead to states in the path simulation relation. The intuition behind the third condition is that all outgoing transitions with valid control inputs from a state must be preserved so that all expected inputs in the state can be captured. Condition four states that, if \( s_1 \) is a wait

![Composition of the garbage collector and its complement.](image-url)
state for a control input, then \( s_2 \) must also be a wait state for the same control input, or \( s_2 \) must lead to a wait state for the same control input.

**Definition 5.10:** A maximal relation over states \((S_p \subseteq S_1 \times S_2)\) is a path simulation relation if \( s_1 \subseteq_p s_2 \) implies:

1. \( T_{s_1}(s') \neq \emptyset \) implies that there exist transitions \( s_1 \xrightarrow{p} s_1, s_2 \xrightarrow{p} s' \) such that \( \mathcal{L}_1 \subseteq \mathcal{L}_2 \) and \( s_1 \subseteq_p s' \).

2. For all transitions \( s_1 \xrightarrow{p} s_1', s_2 \xrightarrow{p} s_2', (\mathcal{L}_1 \subseteq \mathcal{L}_2 \) implies \( s_1' \subseteq_p s_2' \).

3. For all transitions \( s_1 \xrightarrow{p} s_1', (\text{care}_{\mathcal{L}_2} \cap \mathcal{C}_1) \neq \emptyset \) implies there exists a transition \( s_2 \xrightarrow{p} s_2' \) such that \( \mathcal{L}_1 \subseteq \mathcal{L}_2 \) and \( s_1' \subseteq_p s_2' \).

4. If there exist transitions \( s_1 \xrightarrow{p} s_1', s_2 \xrightarrow{p} s_2' \) such that \( s_1 \neq s_1' \) and \( (\text{care}_{\mathcal{L}_2} \cap \mathcal{C}_1) \neq \emptyset \) then (there exists \( s_2 \xrightarrow{p} s_2' \) such that \( \mathcal{L}_1 \subseteq \mathcal{L}_2 \) or (there exists \( s_2 \xrightarrow{p} s_2' \) such that \( \mathcal{L}_1 \subseteq \mathcal{L}_2 \) and \( s_1 \subseteq_p s_2' \)).

**Definition 5.11:** A machine \( M_1 \) is path implemented by machine \( M_2 \) (\( M_1 \subseteq_p M_2 \)) if:

1. The reset state of \( M_1 \) is path simulated by the reset state of \( M_2 \) (\( r_1 \subseteq_p r_2 \)).

2. The final state of \( M_1 \) is path simulated by the final state of \( M_2 \) (\( f_1 \subseteq_p f_2 \)).

We use a simple example to illustrate path implementation. The complement of a garbage collector cycle is transformed into a path implementation. As shown in Figure 5, the wait loop labeled \( \frac{1}{c} \) is removed from the start state. In the resulting machine, the start state has a transition, corresponding to a transition in \( gc \), and the target states for the transitions are also in the path simulation relation. Note that the wait transition in the target state can not be removed because it has an outgoing transition with an active control input.

We will now show that the interaction between a machine and a path implementation of its complement is safe. This means the machines can complete a protocol from the respective start states to the respective final states, with correct control synchronization and data communication at each interaction step. Sequential decomposition involves embedding a path implementation of the complement of a component into the machine from which it is extracted. The theorems in this section provide a formal basis for sequential decomposition.

To prove the above mentioned result we must prove the composability of transition labels in the sequence of interactions between a machine and a path implementation of its complement.

**Lemma 5.1** If \( \mathcal{L}_i, \mathcal{L}_j \) are transition labels and \( \mathcal{L}_i \subseteq \mathcal{L}_j \) then \( \mathcal{L}_i, \mathcal{L}_j \) are composable.

**Proof:** See Appendix

We now prove a general result about the reachability of states in the composition of a machine with a path implementation of its complement.

**Theorem 5.1** Given a machine \( M_1 \) and \( M_2 \) that is path implemented by the complement of \( M_1 \), and a port map \( \pi \) obtained by connecting every port \( p_1 \in P_1 \) with its complementary port \( p_2 \in P_2 \). Let \( M = (M_1 \parallel M_2)_{\pi} \). Then for every state \( \langle s_1, s_2 \rangle \in M \),

\[ \langle s_1, s_2 \rangle \text{ is reachable iff } \exists s_1 \subseteq_p s_1', s_2 \subseteq_p s_2' : (\mathcal{L}_1 \subseteq \mathcal{L}_2 \Rightarrow \mathcal{L}_1 \subseteq \mathcal{L}_2 \text{ and } \langle s_1, s_2 \rangle \rightarrow \langle s_1', s_2' \rangle \in T) \]

where \( s_1 \in M_1, s_2 \in M_2 \) and \( \overline{s}_1 \in \overline{M}_1 \) is the complementary state for \( s_1 \).

**Proof:** Proof by induction on the length of the derivation of states in the construction of \( M \) (see Appendix).

Using theorem 5.1, we can prove that a machine \( M_1 \), composed with a path implementation of its complement \( M_2 \) will result in a safe machine. Since the initial and final states of \( M_1 \parallel M_2 \) correspond to initial and final states in \( M_1 \) and \( M_2 \) separately, we can infer from this theorem that a machine and a path implementation of its complement can complete a protocol. We have the following corollary:

**Theorem 5.2** Given a machine \( M_1 \) and \( M_2 \) that is path implemented by the complement of \( M_1 \), and a port map \( \pi \)
obtained by connecting every port \( p_1 \in P_1 \) with its complementary port \( p_2 \in P_2 \), then \( M = (M_1 \parallel M_2)_s \) is safe.

**Proof:** From the definition of path implementation and complementation, \( f_1 \subseteq_p f_2 \) implies \( (f_1, f_2) \) is reachable in \( M \), and from the definition of path simulation, \( M \) is safe. \( \square \)

6. **EXAMPLE—SCHEME MACHINE DECOMPOSITION**

The Scheme Machine is a special-purpose computer for the Scheme programming language [23]. Its instructions provide base primitives that are used by a run-time system to provide scheme primitives. We now consider the decomposition of a high-level behavioral specification of the Scheme machine into a scheme CPU, an allocator, and a garbage collector (as shown in Figure 1), by sequential decomposition of the scheme machine.

A fragment of the state diagram from the behavioral specification of the Scheme machine is shown in Figure 6. State diagrams provide a graphical representation for abstract behavioral specifications [24].

Our first task will be to decompose the allocator from the scheme machine based on its interface specification. The garbage collection procedure is embedded within the allocator interaction (Figure 6) and we factor it into the allocator as an internal procedure.

The ISL specification for the allocator is shown below:

\[
\text{Allocator}(\text{alloc?}, \text{tag}, \text{size}, \text{avail}, \text{read-avail}, \text{gc-needed})(M) =
\]

\[
\{ \text{await alloc?/T : tag/obj-tag, size/obj-size;}
\]

\[
\text{((read-avail/T until alloc?/F : avail/allocate(obj-tag obj-size M) ;}
\]

\[
\text{[ gc-needed/T until alloc?/F : M(gc(M)) ])}\}
\]

FIGURE 6 Fragment from Scheme Machine Specification

FIGURE 7 Allocator State Diagram.
The state diagram for the allocator is shown in Figure 7. An implementation of an interaction path of the complement of the allocator can be embedded in the scheme machine, in place of the sequence of transitions for allocation interaction in the scheme machine specification (Figure 6). Since the garbage collection procedure is factored into the allocator as an internal procedure, it is not considered in the complement. The source state of the replaced sequence of transitions is merged with the start state of the path implementation [3]. Similarly the target state for the sequence of transitions for allocator interaction is merged with the final state. Figure 8 shows the embedding of a path implementation of the allocator replacing one of the sequence of transitions for allocator interaction shown in Figure 6. Assuming that there is a path from the target state to the source state of the replaced sequence of transitions, the resulting scheme machine description is a path implementation of Allocator, since all transitions in the path can be folded into the target state by hiding all names that are not connected with the allocator. The ports in Allocator (alloc?, tag, size, avail, read-avail, gc-needed) are added to the scheme CPU.

The next step in the derivation is to factor the garbage collector from the allocator. The path implementation of the garbage collector cycle (Figure 5) is then embedded into the allocator by replacing the transition labeled gc(M) (Figure 9). The source state and target state of the replaced sequence of transitions are respectively merged with the start state and final state of the path implementation. The ports in gc (collect, gc-active) are added to the allocator.

We decomposed the Scheme machine into a Scheme CPU and an allocator, and then factored a garbage collector from the allocator. Embedding an implementation of a component into another introduces control synchronization and data communication mechanism between the components for correct interactions between them. The decomposed components can now be synthesized independently.

7. CONCLUSION

Design of a hardware system involves both top-down and bottom-up reasoning. Our emphasis here has been on top-down derivation to promote it as an alternative to the
bottom-up verification techniques. Compared to system synthesis methods, derivation allows a designer greater control over system decomposition and the realization of the components, without the use of special-purpose hardware or restrictions on interactions. We start from an initial abstract description of the system and add details to the description in decomposition steps. Decomposition is accomplished by factoring a procedure to a sequential machine that performs the procedure. With each such transformation step the designer extracts a sequential component from the system and incorporates an implementation of the complement of that component’s interface description into the system. Communication between components is modeled as values over connected ports. Other forms of communication, such as using buffers, shared storage, bidirectionality, must be modeled explicitly. The components in the system can be synthesized independently.

The Scheme machine provided a real example for our decomposition method. We use sequential decomposition to decompose a scheme system into a scheme CPU, an allocator, and a garbage collector with non-trivial interactions between the components. We show that a machine interacting with its complement results is a closed machine that is isomorphic to the original machine, and that a machine interacting with an implementation of its complement results in a safe interaction.

Acknowledgements

We would like to thank the National Science Foundation for its support of this research under grants numbered MIP 89-21842 and MIP 92-08745. We are also grateful to Robert Burger for his help with the examples.

References

DECOMPOSITION OF SEQUENTIAL BEHAVIOR 357


APPENDIX

Proof for Theorem 4.1

Each net of $A/o$ is obtained by connecting an input (alternatively output) port $c$ of $M$ with an output (alternatively input) port $\text{gensym}(c)$ of $M$. From the definition of composition each net in $A/o$ is a port in $M$. Therefore $M$ consists of only output ports and is closed.

The composed machine $M = (M || M)(S)$ has the following structure:

$M = (S, T, (s_i, s_j), T)$

$V = \{\{c, \text{gensym}(c)\} \mid c \in C \}$

$P = \{\}$

$D = \{\}$

$DO = \{\}$

We prove the following two conditions which together imply that $M$ and $M$ are isomorphic:

For $s_i, s_j, s_i', s_j' \in S$

1. $(s_i, s_j) \in S$ if and only if $s_i = s_j$

2. $(s_i, s_j) \in T$ if and only if $s_i = s_j \land s_i' = s_j' \land L_j = \text{Rename}(L_i) \land L_o(p) = L_i(p)$

We will prove the above conditions by induction on the length of the derivation of states in the construction of $M$.

Base Case: $(r, r) \in S$, by the definition of composition.

Inductive Step:

Assume that the above condition is true for all states and transitions derived by $n$ or fewer steps of the construction. Given $(s_i, s_j) \in S \land s_i = s_j \land s_i' = s_j' \land (s_i', s_j') \in T \land s_i' \rightarrow s_j' \in T \land (s_i', s_j') \in T \land s_i' \rightarrow s_j' \in T \land (s_i', s_j') \in T$

By choosing every port $p_i \in \text{care}_{s_i}, \text{care}_{s_j} \cup \text{care}_{s_i} \subseteq \bigcup_{i=1}^{k} [p_i]$

Each net $[p_i]$ consists of two ports $p_i$ and $p_j = \text{gensym}(p_i)$ and $L_o(p) = L_i(p) 

\Rightarrow \forall p, p' \in [p_i], L_i(p) = L_i(p')$

$(s_i, s_j') \rightarrow (s_i', s_j') \in T \land (s_i', s_j') \in T$ for every derivation of length $n + 1$.

To show the converse, assume that $(s_i, s_j') \rightarrow (s_i', s_j') \in T$ is derived in $n + 1$ steps.

From the antecedents of the rule for composition $s_i \rightarrow s_j \in T$ and $s_i' \rightarrow s_j' \in T$ and since $n$ guarantees that $[p_i] = \{p_j, \text{gensym}(p_i)\}$.

We have $s_i \rightarrow s_j \in T, s_i' \rightarrow s_j' \in T$ hence $s_i' = s_j'$.

Proof for Lemma 5.1

Choosing every port $p_i \in \text{care}_{s_i}, \exists p_j \in \text{care}_{s_j} \land \text{care}_{s_i} \subseteq \bigcup_{i=1}^{k} [p_i]$

Each net $[p_i]$ consists of two ports $p_i$ and $p_j$. From the definition of complementation and inclusion, $\forall p_i \in \text{care}_{s_i}, L_i(p) = L_i(p')$

Therefore, $L_i, L_j$ are composable.

Proof for Theorem 5.1

We want to prove the following condition by induction on the length of the derivation of states in the construction of $M$.

$(s_i, s_j) \in S \Leftrightarrow s_i = s_j$

Using lemma 5.1, $(r_1, r_2) \in S$

From the definition of path simulation,

Since, for every transition $r_1 \rightarrow r_2 \in M$, there is a transition $r_1 \rightarrow r_2 \in M$, $r_1 \rightarrow r_2 \in M$.

Using lemma 5.1, $r_1 \rightarrow r_2 \rightarrow (s_1', s_2') \in T$
Inductive Step:

Assume that the above condition is true for all states derived by \( n \) or fewer composition steps. Given \( \langle s_1, s_2 \rangle \) is reachable, by the rule that constructs \( \langle s_1, s_2 \rangle \), we know that there is \( \langle s_{10}, s_{20} \rangle \to\langle s_1, s_2 \rangle \), for some \( \langle s_{10}, s_{20} \rangle \), such that, \( \langle s_{10}, s_{20} \rangle \in M \) is reachable \( s_{10} \to s_1 \in M_1 \) and \( s_{20} \to s_2 \in M_2 \) and \( \mathcal{L}_{10}, \mathcal{L}_{20} \) are composable.

\( s_1 \models_p s_2 \), from induction hypothesis

From the definition of path simulation,

\[
\overline{s_1} \models_p s_2 \Rightarrow \forall \overline{s_1} \models_{\mathcal{F}_1} s_1, s_2 \models_{\mathcal{F}_2} s_2, \mathcal{L}_2 \Rightarrow \overline{s_1} \models_p s_2
\]

Since, for every transition \( \overline{s_1} \models_{\mathcal{F}_1} s_1, s_2 \models_{\mathcal{F}_2} s_2 \in \overline{M}_1 \), there is a transition \( \overline{s_1} \models_{\mathcal{F}_1} s'_1, s'_2 \in \overline{M}_1, \forall \overline{s_1} \models_{\mathcal{F}_1} s_1, s_2 \models_{\mathcal{F}_2} s_2, \mathcal{L}_2 \Rightarrow \overline{s_1} \models_p s_2
\]

Using lemma 5.1,

\[
\mathcal{L}_1 \subseteq \mathcal{L}_2 \Rightarrow \langle s_1, s_2 \rangle \to \langle s'_1, s'_2 \rangle
\]

\( \in T \)

\( \Leftarrow \) : 

To show the converse, \( \langle s_1, s_2 \rangle \to \langle s'_1, s'_2 \rangle \) implies \( \langle s_1, s_2 \rangle \) is reachable in \( M \), from the definition of composition.