Sequential-System Factorization

by

Kamlesh Rath

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science
Indiana University

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Doctoral Committee

Prof. Steven D. Johnson, Ph.D
(Principal Advisor)

Prof. Christopher T. Haynes, Ph.D

Prof. David E. Winkel, Ph.D

Prof. K. Jon Barwise, Ph.D

January 12, 1995

To

Lucy, Baba and Ma

Acknowledgements

I am indebted to my advisor and mentor, Prof. Steven D. Johnson for his guidance and encouragement through the course of my research. His constructive criticism, astute suggestions and inspiring discussions have made this thesis possible. I deeply value the independence he has granted me in undertaking this work. I have always found him approachable, pleasant and helpful. Working with him has been an immensely enjoyable and rewarding experience.

I would like to thank the other members of my committee Prof. David Winkel, Prof. Christopher Haynes and Prof. Jon Barwise for their many useful comments, suggestions and support.

I am specially grateful to Venkatesh Choppella, for our many brain-storming sessions, M. Esen Tuna, for his patience during our collaborative work on Behavior Tables and for giving substance to many of my ideas, Bhaskar Bose for critically evaluating my work and making helpful suggestions, and all of them for being good friends.

I am also thankful to Zheng Zhu, Bob Wehrmeister, Paul Minor, Bob Burger, Shyam Pullela, Dave Boyer and Willie Hunt for the numerous projects we have worked on together and for making my years in the VLSI lab memorable. The administrative and systems staff in the Computer Science department have many a times gone out of their way to help me.

I am deeply grateful to Ignacio and Nora Celis for their friendship throughout my stay in Bloomington, which I will always endear.

I would like to thank NSF: The National Science Foundation, for supporting my research under grants numbered MIP 89-21842 and MIP 92-08745.

I owe a special debt of gratitude to my family for their love and unfailing support. Baba and Ma, thanks for being my constant source of inspiration. Kuni and Jolly, thank you for boosting my morale whenever I needed it. I am grateful to my parents-in-law for their constant encouragement. And finally, I would like to thank my wife, Lucy, for her love, unwavering support, patience and understanding through the ups and downs of the last few years.

Abstract

The success of high-level synthesis methods in reducing design time and formal verification methods in reducing design errors in digital VLSI circuits have opened the way to system-level synthesis and verification. Derivation is a form of formal verification that deals with correct-by-construction reasoning. A set of equivalence preserving transformations are used to derive an implementation from a specification. A key step in derivation is to impose an architectural structure on a behavioral specification by factoring functional behavior into abstract components [40]. These system factorization transformations impose a naive model for synchronization and data communication between components in a system. The thesis of this work is that system factorization must be generalized to support arbitrary interaction protocols between components in a system for derivational methodology to be useful in system-level design.

This dissertation develops a general transformation to decompose a sequential system description into interacting sequential components with non-trivial interaction protocols. Behavior fragments from a system description are encapsulated into a sequential component. Sequential decomposition is used to construct an implementation of interacting sequential components from a higher level of specification, using a description of the interaction of a component with its environment as a parameter. Interface specification language (ISL) is introduced to describe the synchronous interaction of a machine with its environment. The complement of a machine specified in ISL describes the behavior of the machine's environment. Decomposition of sequential components is accomplished by encapsulating parts of a system at the algorithm, process or operation level of granularity into an abstract component with a speci-

fied interaction protocol. An implementation of its complement is then grafted as the interaction stub into the system description to assure correct interaction between components in the system.

Two non-trivial examples are presented to illustrate the use of sequential decomposition in system design. A dynamic memory interface for a formally derived realization of Hunt's FM9001 microprocessor is presented. Sequential decomposition of the DRAM memory interface entails extraction of a DRAM memory object from a system description that incorporates the read/write protocol and accounts for refresh cycles. In the second example, a description of the Scheme system is decomposed into a system of interacting processes that includes a processor, a heap with a stop-and-copy garbage collector and an allocator, and a dynamic RAM based memory sub-system.

Contents

List of Figures

Chapter 1

Introduction

VLSI system design has become increasingly automated over the last decade. This has resulted in significant reductions in design times and errors. Advances in design methods have also made application specific integrated circuits (ASICs) very cost effective and popular. Automatic techniques for logic synthesis of circuits from boolean system descriptions are now being used in the design of most integrated circuits. The synthesis of circuits from high-level descriptions is gaining acceptance in many design domains.

Formal verification of VLSI systems enables designers to prove correctness properties about designs, thereby reducing design errors. With the advent of automatic verification methods there is renewed interest in the industry towards verification to gain a measure of confidence in their designs.

Our approach to design, called *derivation* is a branch of formal verification that deals with correct-by-construction reasoning [38, 40, 42, 39]. A system of equivalence preserving transformations are used to derive an implementation from a specification. We can view such a derivation as a formal proof reflecting a top-down reasoning style. In this respect it should not be viewed as an alternative for deductive (i.e., conventional theorem-prover based) verification but as an alternate mode of reasoning in design [43, 80]. We can also view derivation as a formalization of synthesis, but as a formalization it is more centrally concerned with correctness in reasoning than with automated design.

A specification can have many implementations and a particular derivation chooses one. Hence, the transformations themselves add information to the (accumulating) implementation. Data abstraction in high-level structural descriptions is achieved using transformations based on functional algebra. A transformation called *system factorization* [40] is applied to encapsulate abstract values and operations as simple sequential processes with trivial protocols.

A key aspect of derivation is to impose architectural structure on a behavioral specification. Let us consider the specification of a machine with a number of different arithmetic and logical operations. Many of these operations can be allocated to an ALU using system factorization transformations to abstract arithmetic and logical operations. The derivation can then proceed with the ALU providing part of the structural implementation for the specification.

Similarly, abstract operations on a memory can be encapsulated as a high-level component. The abstract view of memory is realized directly by a static RAM (SRAM) if a certain global timing discipline is followed. But, alternatively, a dynamic RAM (DRAM) would require local protocols to be followed to integrate the memory as a process.

System factorization transformations are strong enough to replace abstract memory *values* in a design, with black-box SRAMs, for example [40], but they need to be further generalized to replace memories with DRAMs. Such a transformation would have to incorporate the read/write protocol and account for refresh cycles. Another example of the problem is described in [43], where the derivation of a microprocessor implementation required moving a naive functional model of memory to a process model with indefinite wait states.

This is also one of the key problems in raising synthesis to the system level. In a related discussion of *interface specification*, Boriello points out that, "the interface component has received limited attention even though it is crucial to integrating the circuit into an environment that will put it to use" [4]. However, while Boriello

develops external interface specifications as a means to guide synthesis, our goal is to use them to guide design decomposition. *Both* sides of the protocol are involved in factoring nontrivial sequential components.

The design of VLSI systems with distributed control requires "glue" logic to interface between system components. First the components must be electrically compatible. The glue logic can be simple to design if the interacting components agree on common protocols for synchronization and data communication. It can also be hard to design for incompatible protocols.

System designers have taken a number of approaches to solve these problems. The first steps were to agree on common voltage and current levels. Logical values based on technologies such as TTL, BiCMOS, ECL were defined for communication between circuit elements. Timing diagrams were invented to report input-output behavior of components. These were then used by system designers to design the interface logic between components in a system. Bus architectures were also used to create standards for interface specifications of components and facilitate designs around a bus.

The design of the interface between components in a system requires suitable partitioning of a system. These are also key steps in the mechanized design of large systems. In a formal design framework, system partitioning and interface design go hand in hand. This dissertation describes a method for partitioning a system specification into interacting components and designing the interface between components.

There are two approaches to interface design. One is to design a separate component that manages the synchronization and data communication with components on both sides of an interaction. The other is to build or choose components with compatible interfaces. The system partitioning method described in this dissertation supports both design approaches and does not restrict a designer to any one of them.

1.1 Sequential Decomposition

Decomposition of system specifications for computer-aided system design is an active topic in system synthesis research. As Gajski [27] points out, system synthesis by design partitioning and interface synthesis must be explored to make synthesis viable for large designs. Synthesis of systems with complex control structures into designs with "monolithic" control units can result in unwieldy circuits. A synthesis tool should have the flexibility to let a designer decompose a system into suitable concurrent 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.

In this dissertation, I develop a general transformation to decompose a sequential system description into interacting sequential components with non-trivial interaction protocols. Behavior fragments from the system description are abstracted into a sequential component. *Interface specification language (ISL)* is introduced to describe the synchronous interaction of a machine with its environment. The complement of a machine specified in ISL describes the behavior of its environment. Decomposition of sequential components 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 grafted as the interaction stub into the system description.

1.2 Dissertation Contributions

The main contribution of this dissertation is to develop a general method for decomposition of system specifications into interacting sequential components. Parts of a system description are encapsulated into sequential processes with non-trivial protocols and the interaction stub for the component are embedded in the system description. The resulting components of a system can then be synthesized independently or

mapped to off-the-shelf components such as dynamic RAMs and floating-point units.

Interface Specification Language (ISL) is used to specify the synchronous interaction protocol of a component in a system orthogonal to its functional behavior. ISL is a formalization of timing diagrams with some extensions such as interleaving to allow for specification of arbitrary protocols. It is a front-end to specify a finite state machine corresponding to the protocol specification. The *complement* of a component gives a description of its environment. Both sides of an interaction must satisfy an *implementation* relation for successful control synchronization and data communication. It is also shown that the *composition* of a machine with an implementation of its complement results in a safe interaction.

1.3 Outline

Chapter 2 describes related works in the areas of research pertaining to this dissertation. The use of formal methods for system design and system-level synthesis are explored in this chapter.

Chapter 3 introduces a language for interface protocol specification. *Interface Specification Language (ISL)* can be used to describe the synchronous interaction of a machine with its environment, orthogonal to its functional behavior. ISL provides a formalization for timing diagrams to describe the control synchronization and data communication between components in a system.

The interpretation of ISL specifications is based on an extended finite state machine model, described in Chapter 4. A visual representation of finite state machines called state diagrams are used to express the interpretation. Chapter 5 explains the minimization, composition and complementation operations on the machines. It also introduces the implementation and path implementation relations over machines.

Chapter 6 describes sequential decomposition using three examples. In the first example, A multiplication procedure is factored out of a factorial machine description and the protocol for interaction with a sequential multiplier is embedded in the facto-

rial machine. The second example describes the decomposition of a DRAM memory sub-system from a microprocessor specification. The steps involved in the derivation of a memory manager unit to manage the protocols between the microprocessor, a DRAM memory bank and a refresh timer are described. The third example illustrates the decomposition of a heap process from the specification of a machine that implements an instruction set for the Scheme functional programming language. In this example, the heap is further decomposed into an allocator process and a garbage collector process.

Chapter 2

Related Research

The research reported in this thesis grew out of earlier work by Johnson [39, 40], Bose [6] and Zhu [79, 78] on transformational methodology for VLSI design. DDD [5, 41] is a transformation system based on Johnson's design derivation algebra. The limitations of the functional transformation framework in dealing with factorizations of sequential components with non-trivial protocols provided the motivation for this work.

In previous derivation exercises, *system factorization* transformations were used to encapsulate parts of a system into abstract objects [41]. These transformations were adequate to factor objects with trivial protocols such as static RAMs, but could not be used for components with non-trivial protocols, e.g. dynamic RAMs. In the derivation of Hunt's FM8501 microprocessor using DDD, Bose derived the entire architecture of the original microprocessor except the memory interface [43]. The memory interface of the FM8501 was injected into Hunt's proof using a "oracle" object that contained a definition of the memory protocol. This could not be incorporated into the description derived by DDD because there was no mechanism to introduce the oracle into the derivation. The goal of this dissertation is to generalize system factorization for components with arbitrary synchronous protocols, such that interface specifications, like the memory interface in the FM8501, can be introduced in the derivation process.

Zhu developed a method to automatically synthesize the interface between two interacting sequential systems by solving the timing constraint equations for the dif-

ferent ports of the systems [81, 78]. His method performs adequately for simple protocols, but can not compose systems with complex interactions like a 4-cycle handshake. The method developed here is more general than Zhu's method in handling complex protocols, but is not automatic.

2.1 Formal Methods

There are three major approaches to formal verification of VLSI systems, automated model checking, theorem proving and process algebra based methods. Formal methods are also being used for design and verification of asynchronous VLSI systems.

2.1.1 Model Checking

Model checking involves extracting models from implementations and verifying invariants, logical and temporal properties about the model in some formal framework. Binary decision diagrams have emerged as a popular modeling paradigm for digital systems [9]. Although model checking has now been used to verify temporal properties of some very large systems [13], it can not be used to verify many other properties that a designer might be interested in. Clarke et.al [14, 12] have used a compositional finite state machine model to verify temporal properties of systems using computational tree logic and binary decision diagram based methods.

Most formal treatments of system decomposition are bottom-up in the sense that they are oriented toward post-design verification. This would include most of the recent research in finite-state machine verification [12]; extensions of FSM models (e.g. [76, 72]) and Petri net theories (e.g. [11]); and model-theoretic work involving process formalisms (e.g. [54, 30]). It is typical that an area of verification research would have this orientation, and also that the top-down view would be better represented in synthesis research. In addition to Boriello's work [4], approaches to scheduling by Ku, Micheli [47] and Nestor et.al [62] have considered protocol-like constraints.

Kurshan uses L-automata with language and process homomorphism [49] to verify reactive systems by stepwise reduction and refinement. He uses a bottom-up model with registers and controllers as processes at the lowest level and constructs complex systems by composing them.

The bottom-up approach models lowest levels of design as processes and composes them for higher levels of description. Each process is modeled as a state machine and composition creates a product machine of the constituent parts. Gopalakrishnan's HOP is a typical example [30]. HOP models processes as state machines with internal behavior and external protocol specifications. HOP specifications are composed using PARCOMP for system level specifications [30]. One problem encountered with this approach is state-space explosion during composition, especially for nondeterministic processes. PARCOMP addresses this problem by pruning extraneous paths in the product machine during construction. As an alternative to bottom-up verification techniques, the methodology described in this dissertation facilitates top-down design by factoring sequential components from designs using transformations.

Dill et.al have used a Büchi automata based model to verify safety and liveness properties using language containment [23]. McMillan and Dill have also modelled timing constraints as min/max constraints and used a generalized branch and bound algorithm to verify the timing specification of connected components [53]. Drusinsky and Harel have used state-charts for hierarchical description of hardware and synthesis of component machines [25]. Holzmann formulated search heuristics to reduce the search space and time for validation of communication protocols [34].

State space explosion is often a problem in bottom-up verification methods using finite state machine models. This occurs in the construction of the product machine. These techniques require the designer to model a system as a network of processes with predefined interactions. The approach described in this dissertation avoids state space explosion by decomposition of an initial abstract description of the system into interacting components. The interaction protocols between components are added to

the system description in decomposition steps.

2.1.2 Deductive Verification

Theorem provers provide a versatile framework for verification and have been used extensively to verify many different kinds of systems, from microprocessors [8] to fault-tolerant systems [3].

Hunt has used the Boyer-Moore Nqthm theorem prover to verify the correctness of the layout of the FM9001 microprocessor against its high-level specification [37, 8]. His earlier verification exercises included the verification of the FM8501 microprocessor [36]. In a similar exercise, the Tamarack microprocessor was also verified by Joyce using HOL [44].

More recently there have been attempts to integrate these different verification methods, using theorem prover based verification for ingenious aspects of system verification and automatic BDD based methods for other verification tasks [67, 68].

The transformational approach to system design is an alternate form of deductive verification where an implementation is derived from a specification using correctness preserving transformations. Since the transformations assure the integrity of the resulting system it obviates the need to verify each implementation against its specification. Each transformation sequence can be considered a proof script. There are limitations to the domain of designs that can be derived using this approach and that can be circumvented by integrating with verification tools [43]. DDD has been used in conjunction with other formal verification tools to verify parts of designs that could not be derived using existing transformations [7, 6]. Sheeran has also used a transformational approach to verification in the Ruby relational algebra [71, 70]. T-Ruby [69] have developed a rewriting tool based on the Ruby algebra for VLSI design derivation.

2.1.3 Process Calculi

Process calculi provide an elegant formal basis for reasoning about interacting sequential systems. Milner's CCS [56, 57] and Pi-calculus [58, 59] have had a great influence on the work in this dissertation. The idea of simulation in this thesis is inspired by the bisimulation relation among processes in CCS and Pi-calculus. Hennessy's work on process algebra [32] and Hoare's CSP [33] have also given me valuable insights into the theoritical aspects of this thesis. These process calculi have been enhanced by many researchers for applications in VLSI design verification. Milne's CIRCAL [55] is one such compositional process algebra with modeling support for buses and clocks. Other bottom-up formal verification methods based on process algebras include [63, 15, 16].

Davie [17] takes a top-down approach to design using verification between specification and implementation steps in CIRCAL. The description of components are composed together for verification with respect to the specification. Contextual constraints, restrictions imposed by a device on its environment are introduced to write partial specifications of a component's environment. The constraints are also used to restrict the target architecture to reduce the complexity of verification [18]. A CIRCAL based transformation to partition a design is mentioned. The designer specifies a component and an algorithm is used to generate the specification of the other component(s) in the design. Davie suggests that this transformation is of "limited usefulness" due to restrictions on it in their formalism. This is similar to the transformational approach described here. The complement operation described in this dissertation constructs the environment in a finite state machine based formalism. It has no restrictions and is central to the decomposition exercise.

2.1.4 Asynchronous Design Methods

Formal methods have been used by many researchers for asynchronous circuit design. The problem of arbitration in asynchronous circuits was first addressed by Muller [61].

Ebergen has also studied the use of arbiters in the decomposition of asynchronous circuits [26].

Martin was the first to use formal design methods to synthesize a working asynchronous microprocessor [1]. His design method for designing self-timed circuits [51] is based on Hoare's CSP [33]. Burns has developed an automatic synthesis method for self-timed circuits based on Martin's work [10].

Dill has also used trace theory in his dissertation to verify asynchronous designs [21]. Nowick and Dill have used petri-nets for verification of asynchronous circutis [24]. Dill et. al have also verified invariant conditions and deadlock avoidance in asynchronous circuits in large systems by down-scaling in the Murφ language [22, 64].

Gopalakrishnan's hopCP is an extension of HOP for formal verification of asynchronous circuits [2, 29, 28]. HOP and hopCP specifications consist of separate protocol and behavior sections representing two orthogonal facets of a process. Similarly, Interface specification language (ISL), developed in this dissertation, is used to specify only the interactions of a machine with its environment. Other facets of a system specification can be expressed in other forms, e.g. behavior tables [66].

The theory developed in this dissertation provides a model and decomposition method for globally synchronous systems. It has many similarities with formal design methods for asynchronous circuits. The notion of "complement of a machine" described here is similar to Dill's description of an "environment" in trace theory [21]. Also, the composition operation described here is similar to Gopalakrishnan's PARCOMP [31].

2.2 System Synthesis

The methodology described in this dissertation explores how a system description can be decomposed into interacting components using a protocol specification with non-trivial control synchronization and data transfer interactions between components. This is different from classical FSM decomposition [20], which assumes tightly-

coupled sub-machines that can share state and input information. Kuehlmann and Bergamaschi have considered system specifications at different levels of abstraction and partitioning of control and data flow graphs to obtain smaller layouts [48]. Yajnik and Ciesielski perform top-down machine decomposition by partitioning outputs and states in state graphs with the objective of performance and area optimization in synthesized PLA circuits [77].

Jóźwiak et.al have developed heuristic methods for simultaneous decompositions of sequential machines into component machines by partitioning the state space, inputs and outputs [46, 45]. 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.

The approach presented in this dissertation enables a designer to decompose machines into logically and functionally distinct components without using heuristics to partition a design based on layout constraints. Synthesis methods are mainly concerned with automatic techniques for optimizing design parameters, such as layout area and critical path length. On the other hand, formal verification methods strive to prove the correctness of a design with respect to its specification. The decomposition method described here can be used to correctly partition a design based on protocol specifications of its components.

Levin describes another synthesis method in which he uses a hierarchical automata model for system specification targeted towards a network of PLAs with memory [50]. 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".

System-level synthesis in the System Architect's Workbench is accomplished by behavioral transformations [75]. Walker and Thomas show transformations on the controller and selector. Transformations to partition a design into processes are also

shown. The processes created using their method have a very simple interaction scheme to transfer data values and control signals using message passing. Their approach can not synthesize components using complex protocols for data transfers and synchronization, e.g. a 4-cycle handshake protocol.

SpecPart [74] partitions algorithm/process grained computations from the SpecChart behavioral specifications. Default protocols are used for interaction between components. The CHOP system-level design partitioner [52] 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 the approach presented in this dissertation, parts of a system at the algorithm/process level or operation level of granularity can be abstracted and the protocol between the components can be incorporated into the components without using any special-purpose modules.

Chapter 3

A Language for Protocol Specification

The word "protocol" has been used in many contexts by different authors. In this dissertation, protocol refers to the synchronous interaction of a process with its environment. This chapter describes *Interface Specification Language (ISL)* to specify the input/output behavior of a sequential machine in its environment. The temporal ordering of input/output actions on the control and data ports is specified with reference to a synchronization signal that is used by the machine and its environment. ISL provides a formal model for timing diagrams with a synchronization signal imposed on it and with extensions for interleaving and repeating timing sequences. We show several examples of protocol specifications in this chapter.

The protocol specification of a process has two components, data and control interaction. 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. Gopalakrishnan's HOP [30] and Zhu's formalism [81] also use a similar convention of defining protocol over separate control and data ports.

We begin with an informal discussion of the syntax and give an interpretation of the language in Chapter 4.

3.1 Syntax

The syntax of the language for interface specification is defined over the following sets: Input control ports (CI), Output control ports (CO), Input data ports (DI) and Output data ports (DO). The ports can range over a set of values (V) which includes the don't care value (#) and first-order terms over other values. As a convention, output ports have an "overline", which is not to be confused with boolean negation. The internal behavior of a process can be annotated in the specification as functions over internal storage elements (R).

The language is built on three kinds of synchronization primitives :

Lock-step : The machine and its environment are synchronized and are prepared for the next transition. No waiting is involved, and there is no need for synchronization signals.

1-way synchronization : One side of the interfaction behaves as the master and the other as a slave. A component or its environment is waiting for the other to finish its computation. The master may take an indeterminate but finite amount of time to finish. The master requires one or more control outputs to signal the transition out of a wait state. The slave needs to wait for the input signal(s) in a wait state.

2-way synchronization : Both sides of a such a synchronization, a component and its environment, are dependent on each other for synchronization. One of them sends a sequence of signals to the other until it receives an acknowledgment signal.

In a 2-way synchronization, one machine signals its transition out of a wait state using one or more control outputs and keeps asserting these outputs until one or more of its input signals are asserted. The other side of this interaction keeps receiving input signals before it acknowledges by asserting output signals.

An informal discussion about actions and expressions in the interface language is given below. We begin with actions that occur in a single step.

1. A data action of the form x/v denotes the data port x and its associated value v. A control action of the form p/v denotes a predicate over the control port p

which is true when the value v is associated with it. Ports with an "overbar" denote output ports.

2. Multiple control actions $(p/v_1, q/v_2, ..., r/v_m)$ and multiple data actions $(x/v_1, y/v_2, ..., z/v_n)$ can occur simultaneously.

3. Atomic actions are of the form $(x : y)$, where x is a set of control actions and y is a set of data actions. Control actions occur as guards for data actions. An *input action* consists of a set of input control actions together with any set of data actions. Similarly an *output action* consists of output control actions and data actions.

4. The actions described above occur in a single step. The following actions span one or more steps.

   (a) (*compute* $\overline{y}$) means output action $\overline{y}$ is performed after some steps. This is used to specify the master side of a 1-way synchronization.

   (b) (*await* y) means a wait loop for input action y. This is used to specify the slave side of a 1-way synchronization.

   (c) (x *before* $\overline{y}$), means that the input action x occurs in one or more consecutive steps before the output action $\overline{y}$ is performed. This is used to specify one side of a 2-way synchronization.

   (d) ($\overline{x}$ *until* y), means output action $\overline{x}$ is performed until input action y occurs. This is used to specify another side of a 2-way synchronization.

5. The expression $(e; a)$ means that the set of actions e is followed by the action a. The unary operator $(;)$ in the expression $(; a)$ is used to denote the initial state and action of the machine.

6. The expression $(e_1 \,[]\, e_2)$ means the choice of actions e_1 or e_2.

7. The expression $(e_1 \mathcal{X} e_2)$ means the sequence of actions in e_1 and e_2 are interleaved. The sets of ports with non don't care values in e_1 and e_2 must be disjoint. Actions in e_1 and e_2 can occur in any sequence as long as their ordering within the expression in preserved.

8. The expression $(e)^*$ means the finite repetition of the sequence of actions e.

9. The complement $(\overline{M})$, composition $(M_1 \parallel M_2)_{\mathcal{N}}$ and Restriction $(Restrict\ (M, P_h))$ operations are discussed in Chapter 4.

The precedence of the operators in descending order is :

$$/ \quad , \quad : \quad \{\ before\ await\ until\ compute\ \} \quad ; \quad [] \quad \mathcal{X}$$

The BNF-style syntax description of the language is given in Figure 3.1. A process is parameterized on the port names occurring in the expression in its definition. Input and output actions are denoted by A_i and A_o respectively.

3.2 Examples

In this section we show a some examples of protocol specifications. The sequential multiplier and dynamic RAM specifications are written from their timing diagrams.

3.2.1 Multiplier

The timing diagram of a sequential multiplier is shown in Figure 3.2. This machine holds the control output $\overline{done}$ until the control input *start* occurs along with the value x on input data port p. This is an example of 1-way synchronization and the multiplier is the slave of its environment during this synchronization. The value y occurs on input data port p in the next transition, an example of lock-step synchronization. After an indeterminate but finite number of steps the control output $\overline{done}$ is asserted with value $x * y$ on the data output port $\overline{o}$, another example of 1-way synchronization

Data Action	$D ::= d_1/v_1, .., d_n/v_n$	where $d \in DI \cup DO$
Control Action	$C ::= c_1/v_1, .., c_n/v_n$	where $c \in CI \cup CO$
Interaction	$A ::= C : D$ $\mid$ *compute* A_o $\mid$ *await* A_i $\mid A_o$ *until* A_i $\mid A_i$ *before* A_o	
Expression	$E ::= ; A \mid E; A \mid E \,[]\, E \mid E \mathcal{X} E$	
Machine	$M ::= E \mid E^* \mid \overline{M} \mid (M \parallel M)_{\mathcal{N}}$ $\mid$ *Restrict* (M, P_h)	where $\mathcal{N}$ ::= port connectivity netlist P_h ::= set of restricted ports

Figure 3.1: BNF-style ISL syntax description

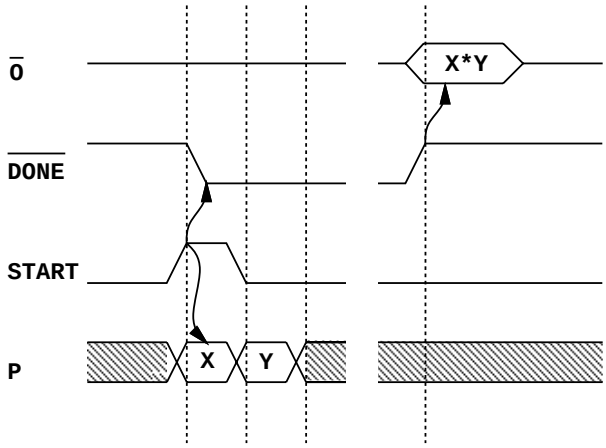

Figure 3.2: Timing diagram for a sequential multiplier

where the multiplier is the master. This sequence of events repeats indefinitely until the machine is turned off or reset.

The ISL protocol specification for *mult* is shown in Figure 3.3. The interpretation of the ISL protocol specifications is described in terms of state diagrams which are based on an extended finite-state machine model discussed in Chapter 4.

3.2.2 Dynamic RAM

In this section we specify the interface of a typical dynamic RAM. To keep the specification brief, we consider the normal modes of operation of the DRAM chip TI-TMS4256.

A timing diagram of the read and CAS-before-RAS refresh cycle, in the normal modes of operation, adapted from the TI-TMS4256 data sheets [73] is shown in Figures 3.4 and 3.5. The write cycle is similar to the read cycle.

The following timing constraints associated with the DRAM are used to choose the reference clock speed for the DRAM specification:

$$\begin{array}{rcrcl} t_1 & \approx & t_a(C) & \geq & 50\text{ns} \\ t_0 + t_1 & \approx & t_a(R) & \geq & 100\text{ns} \\ t_2 & \approx & t_{dis}(CH) & \geq & 30\text{ns} \\ t_3 & \approx & t_w(RH) & \geq & 90\text{ns} \\ t_4 & \approx & t_w(RL) & \geq & 100\text{ns} \end{array}$$

All the above constraints can be satisfied by choosing a clock interval greater than 100ns and letting t_0, t_1, t_2, t_3 and t_4 correspond to a clock interval. A system clock speed of less than 10MHz will meet all the setup and hold timing requirements for normal read, write and CAS-before-RAS refresh cycles. This clock interval is used to normalize all the timing constraints in the timing diagrams to a single synchronizing clock. The worst case memory access time in case of a DRAM is the sum of the read-cycle and refresh-cycle times using a naive interface. The methodology described in

$$mult\ (\text{start}, \overline{\text{done}}, p, \overline{o}) \triangleq [;(\overline{\text{done}}/T\ \ until\ \text{start}/\text{T} : p/x)\ ;\ p/y\ ;\ \ compute\ \overline{\text{done}}/T : \overline{o}/(x * y)]^*$$

Figure 3.3: ISL specification of sequential multiplier

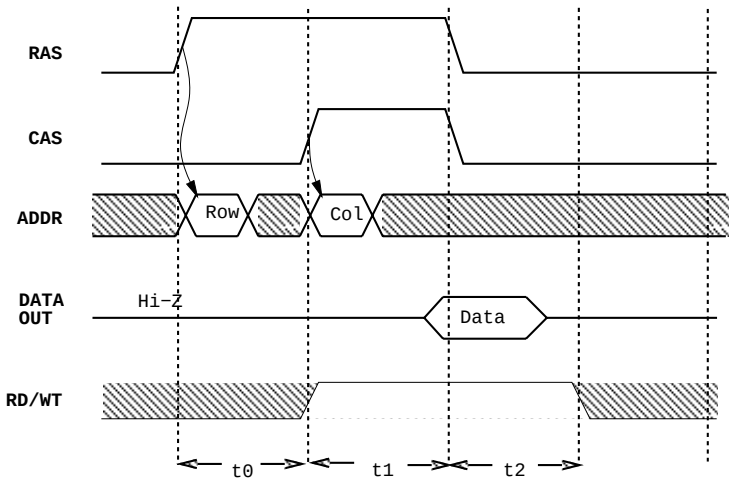

Figure 3.4: Read cycle timing diagram

this dissertation can be used to derive an interface can do better than that with an expected case memory access time.

The ISL specification of the DRAM can be formulated from the timing diagram. The ports in the DRAM are: *ras, cas, addr,* $\overline{dout}$*, din, rw.* $\overline{dout}$, and *din* are the data output and input ports. *addr* is the address port, and *ras, cas* are the row and column address strobe control signals. *rw* is a control signal to select the read cycle when *true* and the write cycle when *false*. The row and column addresses are V_{row}, V_{col} respectively. The internal storage of the DRAM is modeled by the abstract register M. The *read* cycle gets the value from M using an internal operation *(rd* M V_{row} V_{col}*)*. The write cycle updates the memory with the new value V_{write} at the address V_{row}-V_{col}. This internal behavior of the DRAM is annotated in its interface specification. The refresh, read and write cycles form the three choices in the DRAM specification in Figure 3.6.

3.2.3 Garbage Collector

Let us now consider the interface specification of a stop-and-copy garbage collector which will be used in Chapter 6 to perform the decomposition of a scheme system. The garbage collector follows a simple protocol where it waits for *collect* to be asserted and then after a finite interval asserts *gc-active*.

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

3.2.4 Clock Synchronizer

In this section we develop a protocol specification for a fault-tolerant clock synchronizer from formal and informal descriptions in [60]. Four or more such clock synchronizer elements periodically synchronizing their individual clocks using a voting

scheme can provide a fault tolerant clock.

The timing diagram for the clock synchronizer element shown in Figure 3.7 does not provide all required information for a protocol specification of the clock synchronizer. The information that a true value on the input signal t0 can occur before, between or after a true value occurs on output signals $\overline{\mathsf{f1}}$, $\overline{\mathsf{a}dj}$ can not be expressed in the timing diagram. However, it can be expressed using the *interleave* operator in ISL. Also, the outputs $\overline{\mathsf{f1}}$ and $\overline{\mathsf{a}dj}$ must be true one clock tick after T0 goes true. The dependencies between signals where one occurs ≥ 0 clock ticks after another is easily expressed in ISL.

The ISL specification for the clock synchronizer is shown in Figure 3.8. The state machine corresponding to this specification is very large since every interleave operation multiplies the state space. This example demonstrates that ISL provides a compact language for synchronous protocol specifications.

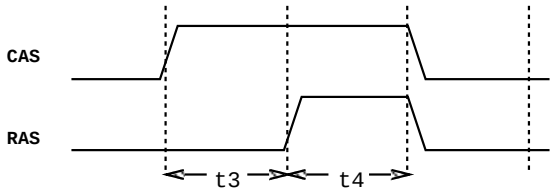

Figure 3.5: Refresh cycle timing diagram

$$
\begin{aligned}
&\mathrm{DRAM}(\mathrm{ras}, \mathrm{cas}, \mathrm{addr}, \overline{\mathrm{dout}}, \mathrm{din}, \mathrm{rw})(M) \triangleq \\
&\qquad ((;\ \mathit{await}\ \mathrm{cas}/T, \mathrm{ras}/F; \mathrm{cas}/T, \mathrm{ras}/T; \mathrm{cas}/F, \mathrm{ras}/F) \\
&\qquad\ \square\ (;\ \mathit{await}\ \mathrm{ras}/T, \mathrm{cas}/F : \mathrm{addr}/V_{\mathrm{row}}; \mathrm{cas}/T, \mathrm{ras}/T, \mathrm{rw}/T : \mathrm{addr}/V_{\mathrm{col}}; \\
&\qquad\qquad \mathrm{cas}/F, \mathrm{ras}/F, \mathrm{rw}/T : \overline{\mathrm{dout}}/(\mathrm{rd}\ M\ V_{\mathrm{row}}\ V_{\mathrm{col}}) \\
&\qquad\ \square\ (;\ \mathit{await}\ \mathrm{ras}/T, \mathrm{cas}/F : \mathrm{addr}/V_{\mathrm{row}}; \mathrm{cas}/T, \mathrm{ras}/T, \mathrm{rw}/F : \mathrm{addr}/V_{\mathrm{col}}; \\
&\qquad\qquad \mathrm{cas}/F, \mathrm{ras}/F, \mathrm{rw}/F : \mathrm{din}/V_{\mathrm{write}}; \{M = (\mathrm{wt}\ M\ V_{\mathrm{row}}\ V_{\mathrm{col}}\ V_{\mathrm{write}})\}))^*
\end{aligned}
$$

Figure 3.6: DRAM protocol specification

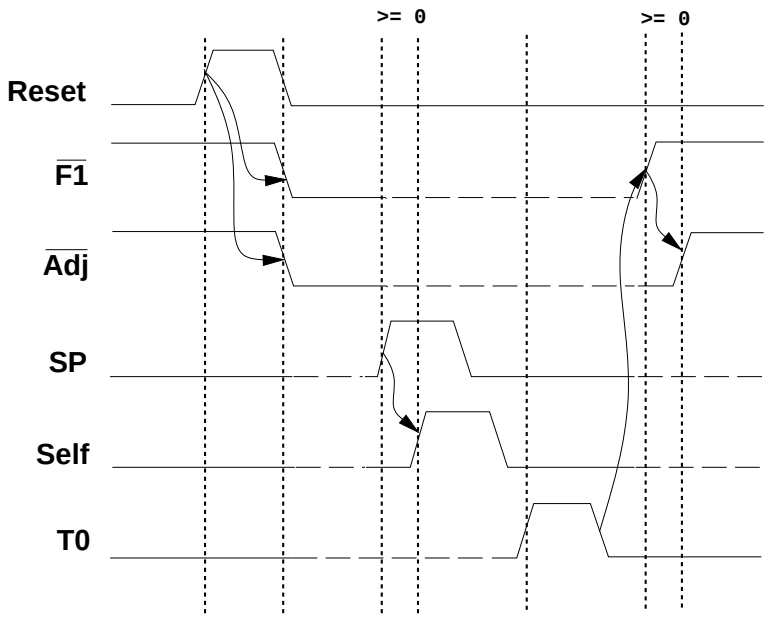

Figure 3.7: Clock synchronizer timing diagram

$$
\begin{aligned}
&\text{CLK-SYNC}(\text{reset}, \text{sp}, \text{self}, \text{t0}, \overline{\text{adj}}, \overline{\text{f1}}) \triangleq \\
&\quad (;\text{reset}/T \ \textit{before}\ \text{reset}/F, \overline{\text{adj}}/F, \text{f1}/F\ ; \\
&\quad ((\ \textit{await}\ \text{sp}/T\ ;\ \textit{await}\ \text{self}/T) \,[]\, (\ \textit{await}\ \text{sp}/T, \text{self}/T))\ ; \\
&\quad (\ \textit{await}\ \text{t0}/T\ ; \overline{\text{adj}}/T, \overline{\text{f1}}/T) \\
&\quad []\,(((\ \textit{await}\ \text{sp}/T\ ;\ \textit{await}\ \text{self}/T) \,[]\, (\ \textit{await}\ \text{sp}/T, \text{self}/T))\ ; \\
&\quad \mathcal{X}(\ \textit{compute}\ \overline{\text{f1}}/T))\ ; (\ \textit{await}\ \text{t0}/T\ ; \overline{\text{adj}}/F, \overline{\text{f1}}/F) \\
&\quad []\,(((\ \textit{await}\ \text{sp}/T\ ;\ \textit{await}\ \text{self}/T) \,[]\, (\ \textit{await}\ \text{sp}/T, \text{self}/T))\ ; \\
&\quad \mathcal{X}((\ \textit{compute}\ \overline{\text{f1}}/T\ ;\ \textit{compute}\ \overline{\text{adj}}/T) \,[]\, (\overline{\text{adj}}/T, \overline{\text{f1}}/T))\ ;\ \textit{await}\ \text{t0}/T))^*
\end{aligned}
$$

Figure 3.8: Clock synchronizer protocol specification

Chapter 4

Interpretation

The semantics of protocol specifications in ISL are based on an extended finite state machine model. This chapter describes the semantic model and the construction of finite state machines from ISL specifications. An operational semantics of the machines is also discussed in this chapter.

4.1 Finite State Machine Model

A machine is a sextuple $M = \langle S, T, r, f, P, R, V \rangle$, where S is the set of states, T is a non-empty set of transitions, r is the start state, f is the final state, P is the set of ports, R is a set of internal registers and V is a domain of values. The sets S, P, R and V are the primitives on which a machine is defined. The set of ports is a union of the disjoint sets of control inputs (CI), control outputs (CO), data inputs (DI), and data outputs (DO). A machine with no input ports is called a *closed* machine.

4.1.1 States

The start state represents the state of the machine after a reset or power-on. 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.

Definition 4.1.1: A state is called a *transit* state if there are no transitions from the state to itself. A machine can stay in a transit state for only one step.

Definition 4.1.2: A state is called a *wait* state if there is at least one transition from the state to itself. The machine takes a transition out of the state based on external conditions (control inputs) or internal conditions (control outputs).

4.1.2 Transitions

Transitions are indicated as: $s_1 \xrightarrow{\mathcal{L}} s_2$, where s_1 is the source state, s_2 is the target state, and $\mathcal{L}$ is the transition label. A label $\mathcal{L}$ is an assignment function $\{\mathcal{L} \mid \mathcal{L} : P \cup R \mapsto V\}$. The set of transitions is a function that maps states and labels to states ($T \subseteq S \times L \times S$), where L is the set of labels.

Definition 4.1.3: The care set for a label $\mathcal{L}$ denotes the set of ports not assigned don't care values by $\mathcal{L}$:

$$care_{\mathcal{L}} = \{p \mid p \in P \wedge \mathcal{L}(p) \neq \#\}$$

4.1.3 State Diagrams

A state diagram is a graphical representation of a finite state machine with nodes representing states and edges representing transitions. The different types of states are depicted using different node symbols : *Transit* - ●, and *Wait* - ⊙. A state of unknown type is depicted as ○. Transitions are shown as labeled edges in the state diagram. The start state is indicated as an incoming edge without any source state.

4.2 Construction

The semantics of a ISL specification is defined in terms of the finite state machine it constructs. We first describe the construction of a state machine from an ISL specification and then give the operational semactics of the state machine.

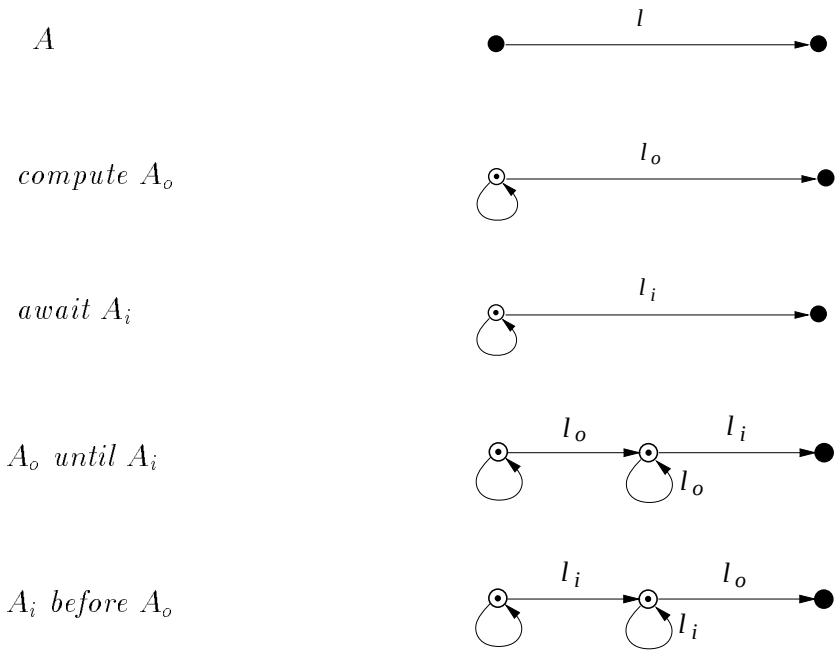

Figure 4.1: Action constructions

4.2.1 Actions

An atomic action of the form $A = \{c_1/v_{c_1}, .., c_m/v_{c_m} : d_1/v_{d_1}, .., d_n/v_{d_n}\}$ denotes a state machine fragment with a transition from one transit state to another transit state with the label l such that

$$l\,(p) \;=\; \begin{array}{ll} v & \text{if } p/v \in A \;\wedge\; p \in P \\ \# & \text{otherwise} \end{array}$$

This transition is taken if $\bigwedge_{i=1}^{m} c_i \in care_l = \mathsf{TRUE}$. The data port d_j has the corresponding value v_j.

Let l_o be the label for an output action A_o, and l_i be the label for an input action A_i. An input action A_i denotes a set of control inputs guards and an output action A_o denotes a set of control output guards. The interpretation of atomic and multi-step actions is shown in Figure 4.1.

The 1-way synchronization primitives are used to specify a master-slave relationship between processes. The syntactic form (*compute* A_o) describes the master process that computes for one or more finite number of steps before the output action labeled A_o. It constructs a transition from a wait state to a transit state (labeled A_o). The syntactic form (*await* A_i) describes the slave process that waits one or more finite steps for an input action A_i.

The syntactic form (A_o *until* A_i) specifies a 2-way synchronization where the output action A_o is performed until the input action A_i is received. It constructs a state machine fragment that starts in a wait state with a transition to another wait state with the label A_o. A transition from the second wait state to a transit state with the label A_i, and a transition from the wait state to itself with label A_o. The syntactic form (A_i *before* A_o) describes a process that receives one or more (finite) input actions A_i before performing the output action A_o.

All the primitive action forms described in Figure 4.1 construct a graph with unique start and final states. These action forms are combined together to form expressions that denote machines. The final state does not have any outgoing transitions to any other state. All the primitive forms also construct machine fragments that have a transit final state. Unlabeled transitions represent hidden actions.

4.2.2 Expressions

A *merge* operation is defined on states and is used in constructing state machine fragments from expressions. States have incoming and outgoing transitions, and can be of the same type or of different types.

Case 1: If both states are of the same type, then this operation creates a single new state of the same type in place of both. All incoming and outgoing transitions of both states are assigned to this state.

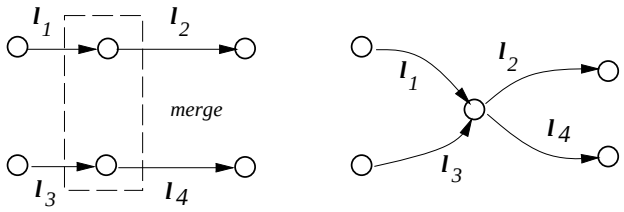

Figure 4.2: Merging states in state machine fragments - Case 1

Case 2: If one state is transit and the other wait, then for each outgoing transition from the wait state, a transition with the same label and same target state is added with the transit state as the source. Since there is always a transition from the wait state to itself, a transition is created from the transit state to the wait state.

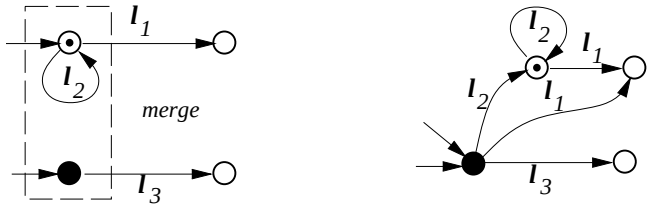

Figure 4.3: Merging states in state machine fragments - Case 2

We now define *asynchronize* and *synchronize* operations on states and then extend these to machines. The *asynchronize* operation on a state adds an unlabeled wait transition to the state. The *synchronize* operation on a state removes any unlabeled wait transitions associated with the state. Figure 4.4 shows these operations on states.

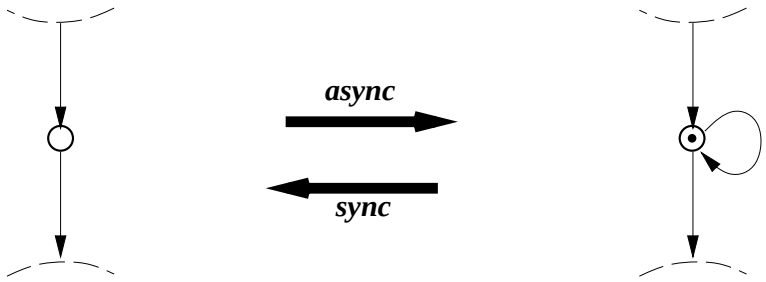

Figure 4.4: Asynchronize and synchronize operations

Let $[D]$ denote the state machine constructed from the action or expression D. The *asynchronize* and *synchronize* operations can be extended to machines by applying the operation to each state in the machine. $[\check{D}]$ denotes the asynchronized machine corresponding to $[D]$ and the synchronized machine corresponding to it is denoted by $[\widehat{D}]$.

An expression in the language begins with ";". This unary operator is used to depict the start state of the machine corresponding to the expression. In the state diagram this is represented by an unlabeled incoming transition with the start state as the target state and no source state. The state machine constructed from expressions also have a unique start and final state like the machine fragments constructed from actions.

Follow:

The binary operator *follows* $(;)$ in the expression $(E;A)$ concatenates $[E]$ with $[A]$. This construction is performed by merging the start state of $[A]$ with the final state of $[E]$.

Choice:

The state machine for the expression $(E_1 [] E_2)$ can be constructed from $[E_1]$ and $[E_2]$ by merging both the start states to form the new start state and both the final states

for the new final state.

Repetition:

The construction of the state machine for the *repetition* operation in the expression (E^*) is done by merging the start and final states of $[E]$.

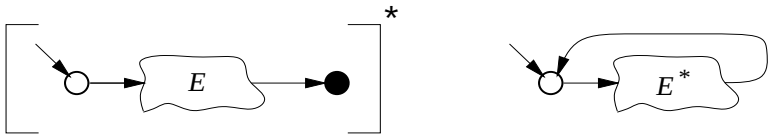

Figure 4.5: Repetition construction

Interleave:

The interleaving of two expressions $(E_1 \mathcal{X} E_2)$ is used to denote any ordering of transitions in $[E_1]$ and $[E_2]$ so that the ordering of transitions in each constituent state machine is preserved.

$$[E_1 \mathcal{X} E_2] \equiv [\check{E}_1] \widehat{\times} [\check{E}_2]$$

where $[E_1] \times [E_2]$ denotes the product machine of $[E_1], [E_2]$

The construction of the product machine is similar to the description in [35].

The start state in the product machine corresponds to the product state formed by the start states of both machines. Each pair of transitions from the start states of both machines result in a *product transition* in the product machine and the pair of target states for the transitions results in the target state of the product transition. The product of two transitions is the union of their assignment functions. Since the set of ports with non don't care values in E_1 and E_2 is disjoint, there are no value conflicts in the product transition. The product machine is constructed inductively using the rules for constructing transitions and states. The final state in the product machine corresponds to the product of the final states of both machines.

4.3 Examples

In this section we show the state machines constructed from the ISL specifications of the sequential multiplier, dynamic RAM and garbage collector described in Chapter 3.

4.3.1 Multiplier

The ISL specification of the sequential multiplier and the state machine constructed from it are shown in Figure 4.6. The multiplier uses a single interaction protocol and uses the port p to get both the values to be multiplied. The start, done signals are used for synchronization.

4.3.2 Dynamic RAM

Figure 4.7 shows the ISL specification for the Dynamic RAM and the state machine constructed from it. The DRAM specification has three protocols, one each for read, write and refresh. The state machine for the DRAM has three corresponding interaction paths from the start to final state. Note that the write protocol modifies the internal state of the memory as indicated by the annotation $\{M = (\text{wt}\ M\ V_{\text{row}}\ V_{\text{col}}\ V_{\text{write}})\}$. The read and refresh protocols do not have annotations and do not change the internal state of the DRAM.

4.3.3 Garbage Collector

Figure 4.8(a) shows the ISL specification and the corresponding state machine for the garbage collector described in Section 3.2.3. The garbage collector also has a single protocol and uses the ports collect, gc-active for control synchronization. The annotation $\{M = \textit{garbage-collect(M)}\}$ abstracts its functional behavior.

$$mult\ (\text{start}, \overline{\text{done}}, p, \overline{o}) \triangleq$$
$$[;(\overline{\text{done}}/T\ \ until\ \text{start/T} : p/x)\ ;\ p/y\ ;\ \ compute\ \overline{\text{done}}/T : \overline{o}/(x * y)]^*$$

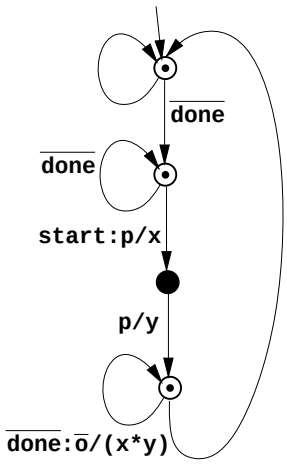

Figure 4.6: Sequential multiplier - ISL specification and state machine

$$\mathrm{DRAM}(\mathrm{ras}, \mathrm{cas}, \mathrm{addr}, \overline{\mathrm{dout}}, \mathrm{din}, \mathrm{rw})(M) \triangleq$$

$$((;\ \mathit{await}\ \mathrm{cas}/T, \mathrm{ras}/F; \mathrm{cas}/T, \mathrm{ras}/T; \mathrm{cas}/F, \mathrm{ras}/F)$$

$$\square\ (;\ \mathit{await}\ \mathrm{ras}/T, \mathrm{cas}/F : \mathrm{addr}/V_{\mathrm{row}}; \mathrm{cas}/T, \mathrm{ras}/T, \mathrm{rw}/T : \mathrm{addr}/V_{\mathrm{col}};$$

$$\mathrm{cas}/F, \mathrm{ras}/F, \mathrm{rw}/T : \overline{\mathrm{dout}}/(\mathrm{rd}\ M\ V_{\mathrm{row}}\ V_{\mathrm{col}})$$

$$\square\ (;\ \mathit{await}\ \mathrm{ras}/T, \mathrm{cas}/F : \mathrm{addr}/V_{\mathrm{row}}; \mathrm{cas}/T, \mathrm{ras}/T, \mathrm{rw}/F : \mathrm{addr}/V_{\mathrm{col}};$$

$$\mathrm{cas}/F, \mathrm{ras}/F, \mathrm{rw}/F : \mathrm{din}/V_{\mathrm{write}}; \{M = (\mathrm{wt}\ M\ V_{\mathrm{row}}\ V_{\mathrm{col}}\ V_{\mathrm{write}})\}))^*$$

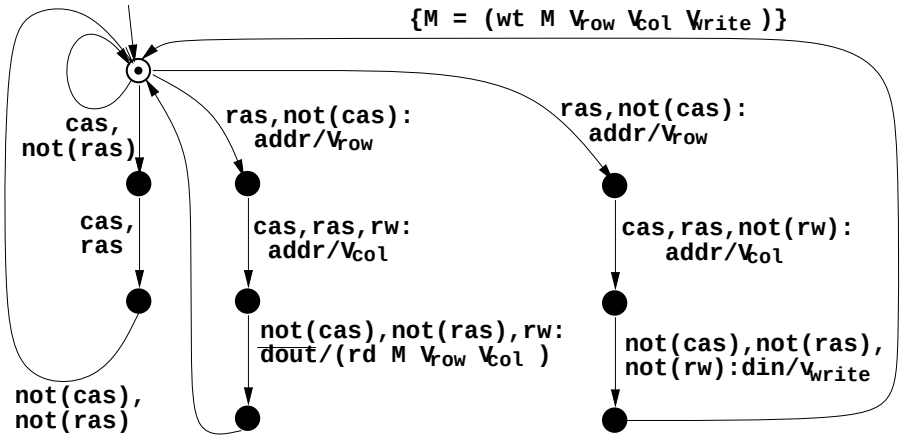

Figure 4.7: DRAM specification and state diagram

$$\mathrm{gc}(\mathrm{collect},\overline{\text{gc-active}})(M) \triangleq$$

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

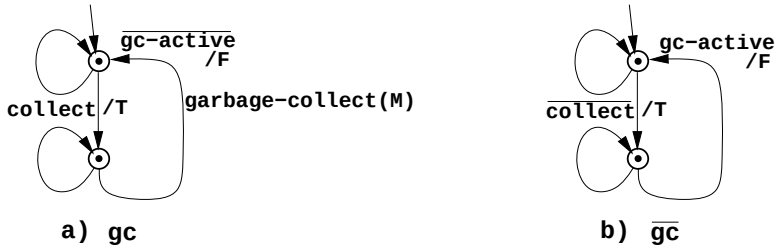

Figure 4.8: Garbage collector (a) and its complement (b)

4.4 Operations on Machines

This section presents some operations on machines. The complement operation is used to create the environment machine. The composition operation constructs a product machine with reachable states and transitions for a given set of port connections. The restriction operation is used to restrict access of certain ports in a machine.

4.4.1 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.

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 state diagram of a machine and its complement have the same graphical structure and there is a one-to-one correspondence between the states and transitions.

Let us define a function "compname" that generates a new input(output) port name for an output(input) port name. In a machine and its complement, the complementary ports (say p and $\overline{p}$ = compname(p)) have the same value in corresponding

transitions.

Definition 4.4.4: Let us now define another function "Rename" that assigns the value of a port to its complementary port in the corresponding transitions of a machine and its complement.

$$\overline{\mathcal{L}}(\overline{p}) = \text{Rename}(\mathcal{L})(\overline{p}) = \mathcal{L}(p)$$

where $\overline{p} \in \overline{P}$ is the new port corresponding to $p \in P$. The "Rename" function creates a new assignment function $\overline{\mathcal{L}}$ corresponding to $\mathcal{L}$.

Definition 4.4.5: Given a machine $M = \langle S, T, r, f, P, R, V \rangle$ where P is its set of port names, the complement machine $\overline{M} = \langle S, \overline{T}, r, f, \overline{P}, \phi, V \rangle$ where $\overline{P}$ is a set of new port names, one for each port in P. The set of transitions

$$\overline{T} = \{s_1 \xrightarrow{\overline{\mathcal{L}}} s_2 \mid s_1 \xrightarrow{\mathcal{L}} s_2 \in T \text{ and } \overline{\mathcal{L}} = \text{Rename}(\mathcal{L})\}$$

The state diagram of the complement of the sequential multiplier machine is shown in Figure 4.9. The complement of the garbage collector is shown in Figure 4.8(b).

4.4.2 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[30]. 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.

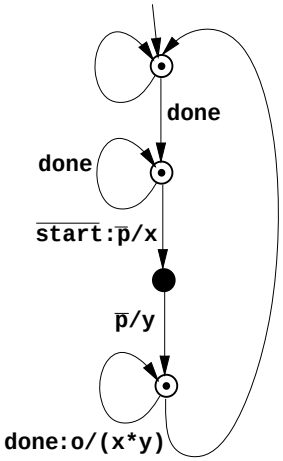

Figure 4.9: Complement of Sequential multiplier

Given machines $M_1 = \langle S_1, T_1, r_1, f_1, P_1, R_1, V_1 \rangle$ and $M_2 = \langle S_2, T_2, r_2, f_2, P_2, R_2, V_2 \rangle$ with disjoint sets of port names $P_1 = CI_1 \cup CO_1 \cup DI_1 \cup DO_1$ and $P_2 = CI_2 \cup CO_2 \cup DI_2 \cup DO_2$, we define the equivalence relation $\mathcal{N}$ on $P_1 \cup P_2$, as follows:

Definition 4.4.6: $p_1 \mathcal{N} p_2$ iff p_1, p_2 are either both control or both data ports and p_1, p_2 are connected. Control and data ports are not allowed to be connected together to keep the formalism simple.

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

Definition 4.4.7: Given machines $M_1 = \langle S_1, T_1, r_1, f_1, P_1, R_1, V_1 \rangle$ and $M_2 = \langle S_2, T_2, r_2, f_2, P_2, R_2, V_2 \rangle$ and the net-list $\mathcal{N}$ between P_1 and P_2, the composed machine $(M_1 \parallel M_2)_{\mathcal{N}}$ is constructively defined by $\langle S, T, r, f, P, R, V \rangle$ where $P = CI \cup CO \cup DI \cup DO$. The sets of port names P_1 and P_2 in M_1 and M_2 are disjoint to avoid conflicts in connecting ports.

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

$CI = \{N \mid N$ contains no control output ports$\}$

$CO = \{N \mid N$ contains at least one control output port$\}$

$DI = \{N \mid N$ contains no data output ports$\}$

$DO = \{N \mid N$ contains at least one data output port$\}$

$$\text{Let} \quad \mathcal{L}_{12}(p) = \begin{cases} \mathcal{L}_1(p) \text{ if } p \in P_1 \\ \mathcal{L}_2(p) \text{ if } p \in P_2 \end{cases}$$

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 $\langle q, s \rangle \xrightarrow{\mathcal{L}} \langle q', s' \rangle \in T$ and $\langle q', s' \rangle \in S$ is reachable iff:
   i. $q \xrightarrow{\mathcal{L}_1} q' \in T_1$ and $s \xrightarrow{\mathcal{L}_2} s' \in T_2$

ii. $\langle q, s \rangle \in S$ is reachable

iii. $\mathcal{L}_1, \mathcal{L}_2$ are *composable*, that is, these exists $j_1.j_2 \ldots j_k$ such that

$$a) \quad care_{\mathcal{L}_1} \cup care_{\mathcal{L}_2} \subseteq \bigcup_{i=1 \text{ to } k} [p_{j_i}]$$

$$b) \quad \forall p, p' \in [p_{j_i}] \ . \ \mathcal{L}_{12}(p) \equiv \mathcal{L}_{12}(p')$$

where the transition label for the composed machine is :
$\mathcal{L}([p]) = \mathcal{L}_{12}(p)$ if $p \in care_{\mathcal{L}_1} \cup care_{\mathcal{L}_2}$, # otherwise

3. $\langle q, s \rangle$ is a transit state if either q or s are both transit states, otherwise it is a wait state.

4.4.3 Restriction

A machine is restricted by eliminating transitions based on certain control ports. This operation is also used to internalize data ports. A list of ports to be hidden $P_h = CI_h \cup CO_h \cup DI_h \cup DO_h$ and a machine $M = \langle S, T, r, f, P, R, V \rangle$, where $P = CI \cup CO \cup DI \cup DO$, must be provided. The transitions with label $\mathcal{L}$ where $(care_{\mathcal{L}} \cap (CI_h \cup CO_h) \neq \phi)$ are removed from the set of transitions, otherwise $\mathcal{L}'(p) = \mathcal{L}(p)$ if $p \in (CI - CI_h) \cup (CO - CO_h)$. The restricted data ports are removed from the transition label, the transitions themselves are not removed. The new transition label in *Restrict* (M, P_h) for the data ports is $\mathcal{L}'_d(p) = \mathcal{L}_d(p)$ if $p \in (DI - DI_h) \cup (DO - DO_h)$.

4.5 Operational Semantics

This section gives a brief description of the abstract syntax of the finite state machine model and an operational semantics for the abstract syntax. The constructions given in sections 4.2 and 4.4 can be used construct a finite state machine that takes a transition on every time step.

$$
\begin{array}{rcl}
M[\tilde{s}] & ::= & [\tilde{p}_i, \widetilde{\mathrm{ci}}_i, \widetilde{\mathrm{co}}_i : \widetilde{\mathrm{di}}_i, \widetilde{\mathrm{do}}_i]_i^K \rightarrow M[\tilde{n}_i] \\
& & \mid \overline{M}[\tilde{s}] \mid \mathit{Restrict}(M[\tilde{s}], p_h) \\
& & \mid (M_1[\tilde{s}_1] \parallel M_2[\tilde{s}_2])_{\mathcal{N}}
\end{array}
$$

Figure 4.10: Abstract syntax for finite state machine model

The abstract syntax of our finite state machine model is shown in Figure 4.10. A machine in its simplest form is a *transition matrix* $[\tilde{p}_i, \widetilde{\mathrm{ci}}_i, \widetilde{\mathrm{co}}_i : \widetilde{\mathrm{di}}_i, \widetilde{\mathrm{do}}_i]_i^K$ with K rows and each row subscripted by its row index i. Each row describes a guarded transition of the form $\tilde{p}_i, \widetilde{\mathrm{ci}}_i, \widetilde{\mathrm{co}}_i : \widetilde{\mathrm{di}}_i, \widetilde{\mathrm{do}}_i \rightarrow M[\tilde{n}_i]$ where $\tilde{p}_i$ is the present-state vector, $\widetilde{\mathrm{ci}}_i$ is the vector of control inputs, $\widetilde{\mathrm{co}}_i$ is the vector of control outputs, $\widetilde{\mathrm{di}}_i$ is the vector of data inputs, $\widetilde{\mathrm{do}}_i$ is the vector of data outputs and $\tilde{n}_i$ is the next-state vector. The state vector represents all the internal registers in the machine. The present-state vector $\tilde{p}_i$ denotes all the values in the internal registers during the transition and the next-state vector $\tilde{n}_i$ denotes all the values that will be assigned to the internal registers during the transition.

A machine can also be expressed as a complement, restriction or composition of machines. The restriction operation requires a set of ports to be hidden (P_h) and the composition operation requires a net-list of connected ports ($\mathcal{N}$).

The operational semantic rules are defined over the abstract syntax as in [65, 56, 30]. We consider machine transitions at a time step (t) and define the relation given below by structural induction.

$$
M[\tilde{p}_i](t) \xrightarrow{\widetilde{\mathrm{ci}}_i(t), \widetilde{\mathrm{co}}_i(t) : \widetilde{\mathrm{di}}_i(t), \widetilde{\mathrm{do}}_i(t)} M[\tilde{n}_i](t+1)
$$

where $\widetilde{\mathrm{ci}}_i(t), \widetilde{\mathrm{co}}_i(t), \widetilde{\mathrm{di}}_i(t), \widetilde{\mathrm{do}}_i(t)$ are the possible control and data actions of $M[\tilde{p}_i]$ at time t. We omit t and $(t+1)$ in the rest of the discussion.

Deterministic Choices: A transition in a machine is chosen to be executed if the values of the present state, control inputs and outputs satisfy the guards in the

transition. The corresponding data inputs and outputs actions are executed and the machine enters the next state.

$$([\tilde{p}_i, \widetilde{\mathrm{ci}}_i, \widetilde{\mathrm{co}}_i : \widetilde{\mathrm{di}}_i, \widetilde{\mathrm{do}}_i]_i^K \rightarrow M[\tilde{n}_i]) \xrightarrow{\tilde{\mathrm{p}}_i, \widetilde{\mathrm{ci}}_i, \widetilde{\mathrm{co}}_i : \widetilde{\mathrm{di}}_i, \widetilde{\mathrm{do}}_i} M[\tilde{n}_i]$$

Complementation: The complement of a machine has its input and output ports reversed. The execution of a machine and its complement are identical except input actions are now output actions and vice versa.

$$\frac{M[\tilde{p}_i] \xrightarrow{\tilde{\mathrm{p}}_i, \widetilde{\mathrm{ci}}_i, \widetilde{\mathrm{co}}_i : \widetilde{\mathrm{di}}_i, \widetilde{\mathrm{do}}_i} M[\tilde{n}_i]}{\overline{M}[\tilde{p}_i] \xrightarrow{\tilde{\mathrm{p}}_i, \widetilde{\mathrm{co}}_i, \widetilde{\mathrm{ci}}_i : \widetilde{\mathrm{do}}_i, \widetilde{\mathrm{di}}_i} \overline{M}[\tilde{n}_i]}$$

Note in the rule above that the positions of the control(data) input and output ports have been swapped between the machine and its complement.

Restriction: These rules capture the result of restricting control ports and internalizing data ports. Restricting control input or output ports results in the pruning of transitions where the restricted control ports have non don't care values.

Control:

$$\frac{M[\tilde{p}_i] \xrightarrow{\tilde{\mathrm{p}}_i, \widetilde{\mathrm{ci}}_i, \widetilde{\mathrm{co}}_i : \widetilde{\mathrm{di}}_i, \widetilde{\mathrm{do}}_i} M[\tilde{n}_i]}{\mathit{Restrict}(M[\tilde{p}_i], c_h) \xrightarrow{\tilde{\mathrm{p}}_i, \widetilde{\mathrm{ci}}_i, \widetilde{\mathrm{co}}_i : \widetilde{\mathrm{di}}_i, \widetilde{\mathrm{do}}_i} \mathit{Restrict}(M[\tilde{n}_i], c_h)} \mathcal{L}(c_h) \neq \# \text{ or } c_h \notin CI \cup CO$$

Restricted data ports are internalized and are not visible outside. The data value associated with the restricted port is removed from the vectors of data input and output values ($\widetilde{\mathrm{di}}_i \backslash d_h, \widetilde{\mathrm{do}}_i \backslash d_h$). No transitions are pruned.

Data:

$$\frac{M[\tilde{p}_i] \xrightarrow{\tilde{\mathrm{p}}_i, \widetilde{\mathrm{ci}}_i, \widetilde{\mathrm{co}}_i : \widetilde{\mathrm{di}}_i, \widetilde{\mathrm{do}}_i} M[\tilde{n}_i]}{\mathit{Restrict}(M[\tilde{p}_i], d_h) \xrightarrow{\tilde{\mathrm{p}}_i, \widetilde{\mathrm{ci}}_i, \widetilde{\mathrm{co}}_i : \widetilde{\mathrm{di}}_{i \backslash d_h}, \widetilde{\mathrm{do}}_{i \backslash d_h}} \mathit{Restrict}(M[\tilde{n}_i], d_h)} d_h \in DI \cup DO$$

Composition: We now consider the composition of two machines with respect to a net-list. The rule below computes the behavior of $(M_1[\widetilde{p1}] \parallel M_2[\widetilde{p2}])_{\mathcal{N}}$ from the behavior of $M_1[\widetilde{p1}]$ and $M_2[\widetilde{p2}]$.

Let $\mathcal{L}_1, \mathcal{L}_2$ be $\{\widetilde{\mathrm{p1}}, \widetilde{\mathrm{ci1}}, \widetilde{\mathrm{co1}} : \widetilde{\mathrm{di1}}, \widetilde{\mathrm{do1}}\}, \{\widetilde{\mathrm{p2}}, \widetilde{\mathrm{ci2}}, \widetilde{\mathrm{co2}} : \widetilde{\mathrm{di2}}, \widetilde{\mathrm{do2}}\}$ respectively. Also, let $[p]$ be a net in $\mathcal{N}$ and P_1, P_2 be the set of ports in M_1, M_2 respectively.

$$\frac{M_1[\widetilde{p1}] \xrightarrow{\mathcal{L}_1} M_1[\widetilde{n1}]\ ,\ M_2[\widetilde{p2}] \xrightarrow{\mathcal{L}_2} M_2[\widetilde{n2}]}{(M_1[\widetilde{p1}] \parallel M_2[\widetilde{p2}])_{\mathcal{N}} \xrightarrow{\mathcal{L}} (M_1[\widetilde{n1}] \parallel M_2[\widetilde{n2}])_{\mathcal{N}}} \mathcal{L}_1, \mathcal{L}_2 \textit{ are composable}$$

where $\mathcal{L}([p]) = \mathcal{L}_1(p)$ if $p \in P_1$ or $\mathcal{L}_2(p)$ if $p \in P_2$ (from definition 4.4.7).

Chapter 5

Sequential Decomposition

This chapter describes isomorphism and implementation relations over machines. A notion of safety in a composed machine is introduced and it is shown that the composition of a machine with a path implementation of its complement results in a safe interaction between the machines. This chapter also elaborates on the main contribution of this dissertation, a method of decomposition of sequential components from machine descriptions.

5.1 Machine Isomorphism

This section describes an isomorphism relation between machines based on their isomorphic sets of states and transitions. This is a strong relation between machines and it means that state machines have the same topology. The machines also exhibit the same protocol behavior modulo the names and values on their ports.

Definition 5.1.8: Two machines are graph-isomorphic if they have isomorphic sets of states and transitions.

As described in Section 4.1, a machine with no input ports is called a closed machine. It is shown here that a machine composed with its complement results is a closed machine that is graph-isomorphic to the original machine. This isomorphism shows that the temporal characterstics of a machine are preserved. The validity of the interaction between a component of a system and the rest of the system can be assured

by incorporating the complement of a sequential component into the specification of the rest of the system. The intuition behind this result is that a component in a system behaves predictably and completes its interactions if its environment behaves as its complement.

Let us now 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 5.1.1 *Given a minimal deterministic machine* $M = \langle S, T, r, f, P, R, V \rangle$ *where* $P = \mathrm{CI} \cup \mathrm{CO} \cup \mathrm{DI} \cup \mathrm{DO}$ *and its complement machine* $\overline{M} = \langle S, \overline{T}, r, f, \overline{P}, \phi, V \rangle$ *and* $\mathcal{N}_0$ *is the port map obtained by connecting every port* $p \in P$ *with its complementary port* $p' \in \overline{P}$*. Let* $M_0 = (M \parallel \overline{M})_{\mathcal{N}_0}$*. 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. □

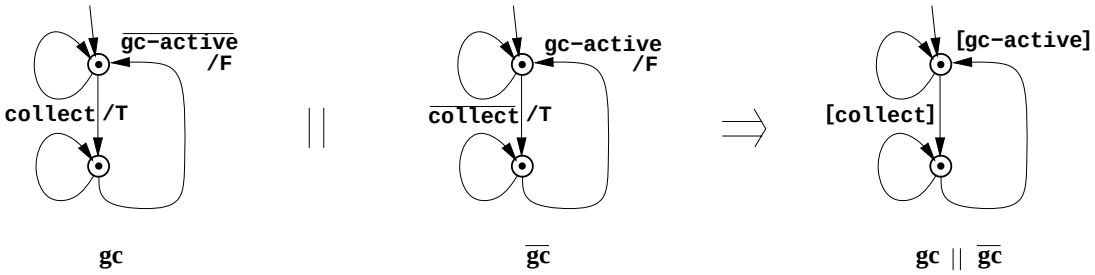

Figure 5.1: Composition of the garbage collector and its complement

Theorem 5.1.1 proves that a machine composed with its complement results in a closed machine graph-isomorphic to itself. There is a one-to-one mapping between the states and transitions in M and M_0. Thus there is no state space explosion in the composed machine. The operational behavior of the composed machine M_0 is the same as M, but M_0 is closed and self contained. Figure 5.1 shows the construction

of the machine composed using the garbage collector and its complement with every port connected with its complementary port. As shown in the example, the composed machine $(\text{gc} \parallel \overline{\text{gc}})_{\mathcal{N}_0}$ is isomorphic to the original gc, and has no input ports.

5.2 Implementation

Let us now explore relations over machines based on their input/output behavior. The implementation relation described in this section is not based on the structure of the state graph, but on the sequence of observable states and transitions that the machines go through during interaction. The transitions of a machine and the values associated with the ports in the machine for each transition are key to the implementation relation. 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* ($\sqsubseteq$) relation and a weak *path implementation* ($\sqsubseteq_{\text{p}}$) relation.

The relation between two machines is based on the correspondence between their port sets. 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 are type preserving in a broad sense so that the values on these ports can be compared.

Definition 5.2.9: The *inclusion* relation over transition labels with respect to a port map is defined as:

$$\mathcal{L}_1 \preceq \mathcal{L}_2 \iff \forall p_1 \in care_{\mathcal{L}_1} \,.\, \exists p_2 \in care_{\mathcal{L}_2} \,.\, p_1 \mapsto p_2 \ \wedge \ \mathcal{L}_1(p_1) = \mathcal{L}_2(p_2)$$

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

Definition 5.2.10: A relation over states is a *simulation* relation if $s_1 \sqsubset s_2$ implies:

$$\forall s_1 \xrightarrow{\mathcal{L}_1} s_1' \,.\, \exists s_2 \xrightarrow{\mathcal{L}_2} s_2' \,.\, \mathcal{L}_1 \preceq \mathcal{L}_2 \ \wedge \ s_1' \sqsubset s_2'$$

A machine M_1 is implemented by ($\sqsubseteq$) 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 5.2.11: The relation M_1 *implementated by* M_2 ($M_1 \sqsubseteq M_2$) holds iff:

$r_1 \sqsubset r_2$ *and* $(\forall s_1 \in S_1 \,.\, \exists s_2 \in S_2 \,.\, s_1 \sqsubset s_2)$.

5.3 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.

A *path implementation* of a machine implements one of many protocols of that 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 *plus* and *divide*. Such a device would then have distinct complements for the plus protocol and the divide protocol - we call these individual partial complements *path complements* because they characterize different paths through the component 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, $\mathcal{S}_p \subseteq S_1 \times S_2$, where S_1, S_2 are the sets of states in the two machines. $s_1 \sqsubset_{\mathrm{p}} s_2$ implies four conditions (Definition 5.3.12, below). 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 this 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 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 5.3.12: A maximal relation over states ($\mathcal{S}_p \subseteq S_1 \times S_2$) is a *path simulation* relation if $s_1 \sqsubset_{\mathrm{p}} s_2$ implies:

1. $T_1(s_1) \neq \phi \Rightarrow \exists s_1 \xrightarrow{\mathcal{L}_1} s_1', s_2 \xrightarrow{\mathcal{L}_2} s_2' \,.\, \mathcal{L}_1 \preceq \mathcal{L}_2 \ \wedge \ s_1' \sqsubset_{\mathrm{p}} s_2'$
2. $\forall s_1 \xrightarrow{\mathcal{L}_1} s_1', s_2 \xrightarrow{\mathcal{L}_2} s_2' \,.\, (\mathcal{L}_1 \preceq \mathcal{L}_2 \Rightarrow s_1' \sqsubset_{\mathrm{p}} s_2')$
3. $\forall s_1 \xrightarrow{\mathcal{L}_1} s_1' \,.\, (care_{\mathcal{L}_1} \cap CI_1) \neq \phi \ \Rightarrow \ \exists s_2 \xrightarrow{\mathcal{L}_2} s_2' \,.\, \mathcal{L}_1 \preceq \mathcal{L}_2 \ \wedge \ s_1' \sqsubset_{\mathrm{p}} s_2'$
4. $\exists s_1 \xrightarrow{\mathcal{L}_{1_1}} s_1, s_1 \xrightarrow{\mathcal{L}_{1_2}} s_1' \,.\, s_1 \neq s_1' \ \wedge \ (care_{\mathcal{L}_{1_2}} \cap CI_1) \neq \phi$
   $\Rightarrow \ (\exists s_2 \xrightarrow{\mathcal{L}_{2_2}} s_2' \,.\, s_2 \neq s_2' \ \wedge \ \mathcal{L}_{1_2} \preceq \mathcal{L}_{2_2} \ \wedge \ s_1' \sqsubset_{\mathrm{p}} s_2') \ \wedge$
   $((\exists s_2 \xrightarrow{\mathcal{L}_{2_1}} s_2 \,.\, \mathcal{L}_{1_1} \preceq \mathcal{L}_{2_1}) \ \vee \ (\exists s_2 \xrightarrow{\mathcal{L}_{2_3}} s_k \,.\, \mathcal{L}_{1_1} \preceq \mathcal{L}_{2_3} \ \wedge \ s_1 \sqsubset_{\mathrm{p}} s_k))$

The *path simulation* relation over states is now extended to machines. As defined below, the unique start/reset states and the unique final states of two machines determine if the machines are in the *path implementation* relation.

Definition 5.3.13:

A machine M_1 is *path implemented by* machine M_2 ($M_1 \sqsubseteq_{\mathrm{p}} M_2$) if

1. The start state of M_1 is path simulated by the start state of M_2 ($r_1 \sqsubset_{\mathrm{p}} r_2$).
2. The final state of M_1 is path simulated by the final state of M_2 ($f_1 \sqsubset_{\mathrm{p}} f_2$).

Let us now consider a simple example to illustrate path implementation. The complement of a garbage collector cycle is transformed into a path implementation. As shown in Figure 5.2, the wait loop labeled $\boxed{1}$ is removed from the start state. In

the resulting machine, the start state has a transition, corresponding to a transition in $\overline{\text{gc}}$, 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.

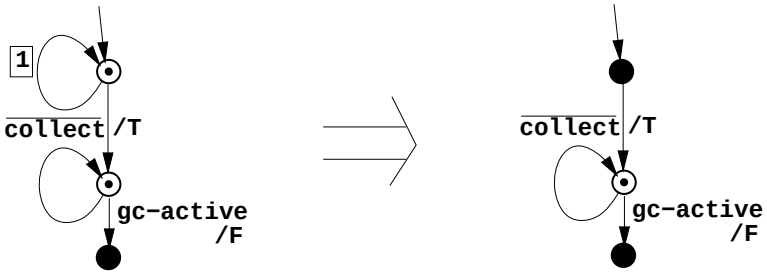

Figure 5.2: Path implementation of $\overline{\text{gc}}$

5.4 Decomposition

Given a specification of a sequential process at a high-level of abstraction, we would like to decompose the process in two or more sub-processes. A procedure $f(e)$ in the specification can be instantiated as a subprocess P_f. The goal here is to incorporate an implementation of the complementary process $\overline{P_f}$ as an interaction stub into the original process.

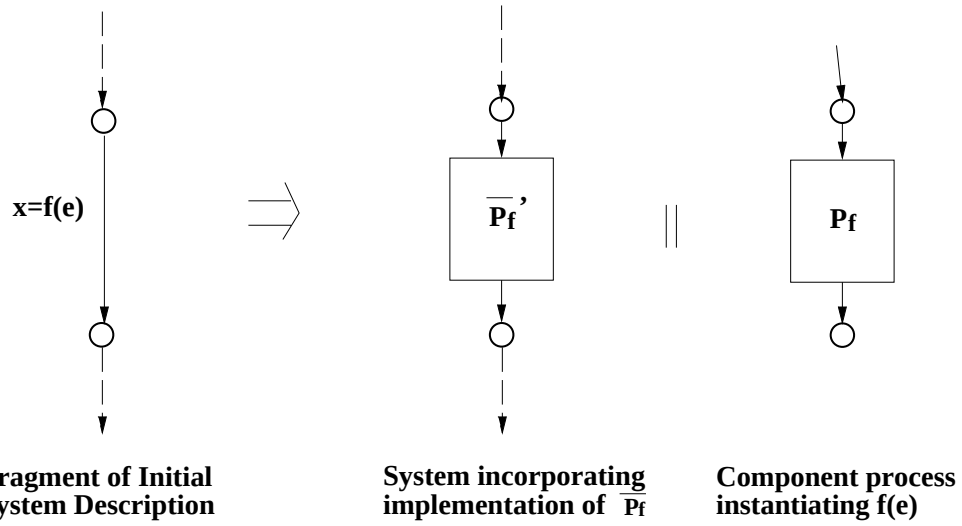

Figure 5.3: Top-down decomposition of sequential component

Figure 5.3 illustrates the method of decomposition of a sequential component from a system description. An operation in the specification at the procedure/function level of granularity can be factored as a sequential component in a peer relationship. The protocol specification of the sequential component describes all possible interactions it can engage in. Its complement describes all possible behaviors it can expect from its environment. The component could implement many different operations one or some of which may correspond to the factored operation. A protocol stub from the complement of the component that implements the relevant operation is grafted into the specification in place of the operation. The grafted protocol stub must be a *path implementation* of the complement of the component for proper interaction between them.

In Figure 5.3, $f(e)$ is instantiated in the process P_f. The process $\overline{P_f}'$ is an implementation of the complement process ($\overline{P_f}$) of P_f. The transitions containing the procedure $f(e)$ is replaced with $\overline{P_f}'$ by merging the start state with the source state of the transition and the final state with the target state of the transition. Any other operations in the replaced transition are scheduled in the earliest possible transition

in $\overline{P_f}'$ that does not go from a state to itself, such that the register value dependencies in the original machine are preserved.

The parameters in the procedure $f(e)$ are now passed between the process P_f and its protocol stub $\overline{P_f}'$ as values assigned to specific ports on specific transitions. The implementation relation between the complement of P_f and $\overline{P_f}'$ assures that the parameter values and results are passed properly between them.

If more than one procedure in a transition are decomposed into sequential processes, then the protocol stub grafted in place of the transition must be a path implementation of the complements of all the sequential processes that it interacts with. This ensures that the collateral protocols with the different sequential processes can be executed properly.

5.5 Safe Interaction

The decomposition of a system into processes introduces synchronization between them. The correctness of the decomposition depends on functional correctness of the component and the interaction between the machines. In this thesis, we discuss the correctness of the interaction protocol between machines, assuming the functional correctness of the decomposed component.

If the final states of two interacting machines are reachable then we say that the composition of the two machines is *safe*. The composition operation is defined so that the composed machine is built inductively starting from the start states of the constituent machines and then considering pairs of transitions from reachable states that can interact with each other. All states and transitions that are not reachable by interaction are pruned. Both interacting machines must follow a correct protocol to reach their final states from their start/reset 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 5.5.14: The composed machine $(M_1 \parallel M_2)_{\mathcal{N}}$ is considered *safe* if the state $\langle f_1, f_2 \rangle$ is reachable, *unsafe* otherwise.

Now let us prove that the interaction between a machine and any path implementation of its complement is safe. This means a machine interacting with any path implementation of it complement can complete a protocol from their respective start states to their respective final states.

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

Lemma 5.5.1 *If $\mathcal{L}_i, \mathcal{L}_j$ are transition labels and $\overline{\mathcal{L}_i} \preceq \mathcal{L}_j$, 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 5.5.1 *Given a machine M_1 and M_2 that is path implemented by the complement of M_1, and a port map $\mathcal{N}$ obtained by connecting every port $p_1 \in P_1$ with its complementary port $p_2 \in P_2$. Let $M = (M_1 \parallel M_2)_{\mathcal{N}}$. Then for every state $\langle s_1, s_2 \rangle \in M$, $\langle s_1, s_2 \rangle$ is reachable iff*

$$a) \quad \overline{s_1} \sqsubset_{\mathrm{p}} s_2$$

$$b) \quad \forall s_1 \xrightarrow{\mathcal{L}_1} s_1', s_2 \xrightarrow{\mathcal{L}_2} s_2' \,.\, (\overline{\mathcal{L}_1} \preceq \mathcal{L}_2 \;\Rightarrow\; \overline{s_1'} \sqsubset_{\mathrm{p}} s_2' \;\wedge\; \langle s_1, s_2 \rangle \rightarrow \langle s_1', s_2' \rangle \in T)$$

where T is the set of transitions in M; $s_1 \in M_1, s_2 \in M_2$ 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). □

Theorem 5.5.1 states that in the described composition two states can only be composed if one is a complement of a state which is in a path simulation relation with the other. Also, all composable pairs of transitions (defined in Lemma 5.5.1) from these states and their target states are part of the composed machine.

We can now prove using theorem 5.5.1 that a machine M_1 composed with any 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 5.5.2 *Given a machine M_1 and M_2 that is path implemented by the complement of M_1, and a port map $\mathcal{N}$ obtained by connecting every port $p_1 \in P_1$ with its complementary port $p_2 \in P_2$, then $M = (M_1 \parallel M_2)_{\mathcal{N}}$ is safe.*

Proof: From the definition of *path implementation* and *complementation*, $\overline{f_1} \sqsubset_{\mathrm{p}} f_2$ implies $\langle f_1, f_2 \rangle$ is reachable in M, and from the definition of *path simulation*, M is safe. □

Theorems 5.5.1 and 5.5.2 provide the theoritical basis for sequential decomposition. The interaction between sequential components in a decomposed system is *safe* provided the connections between the machines are proper. This theorem does not state anything about the functional correctness of the decomposed component. The intuition behind these results is that a machine can have a useful interaction with its environment if it gets the right values when it expects them and outputs the right values when its environment expects them. Moreover, such an interaction is considered safe if it completes a protocol, assuming the functional correctness of the components. Further discussion about the functional correctness of the components is deferred to Chapter 7. This result is central to the decomposition method described in this dissertation and the next chapter illustrates its applications.

Chapter 6

Examples

This chapter illustrates the use of sequential decomposition in system design using three examples. A component can be decomposed from a system specification by incorporating the appropriate *path implementation* of its complement into the description of the rest of the system.

The first example illustrates the steps involved in the decomposition of a sequential multiplier from a factorial machine. This is a simple example where the multiplier has a single protocol and it communicates exclusively with the factorial machine.

The second example shows the decomposition of a Scheme system description into a system of interacting processes. The system includes a processor, a heap with a stop-and-copy garbage collector and an allocator, and a dynamic RAM based memory sub-system. In the current realization of the Scheme system, each component was individually derived from a separate specification. But, the interactions between the components were not derived. This example describes the decomposition of the garbage collector and the memory allocator from an initial Scheme system description based on their protocol specifications. Also, the interactions between the decomposed components are *safe*. This example brings out the ability of the decomposition method to deal with embedded protocols where the allocator interaction subsumes the garbage collector interaction.

The third example describes the application of sequential decomposition to derive a dynamic memory interface for a formally derived realization of the Nqthm FM9001

microprocessor specification [37], called DDD-FM9001 [6]. Sequential decomposition of the DRAM memory interface entails extraction of a DRAM memory object from a system description that incorporates the read/write protocol and accounts for refresh cycles. The current realization of the DDD-FM9001 processor [6] does not support a DRAM interface because of the limitations in deriving an interface with non-trivial protocols. This example shows how sequential decomposition is used to derive interfaces with non-trivial collateral protocols, such as the memory interface that must simultaneously interact with the dynamic memory and the processor.

6.1 Factorial Machine Decomposition

Consider the example of a factorial machine *fac* (Figure 6.1). This specification shows the internal behavior of *fac*, e.g. internal registers u, v, and internal conditions $[u = 0], [u \neq 0]$.

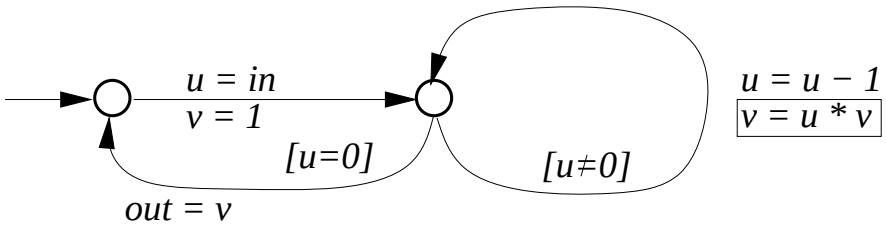

Figure 6.1: Factorial Machine State Diagram

To abstract the procedure $v = u * v$ and use the multiplier machine specified in Figure 3.3, the factorial machine should incorporate an *implementation* of the complement machine $\overline{mult}$ (Figure 4.9). The assumption here is that *mult* performs multiplication. The following steps are involved in modifying *fac*:

1. Add all the control and data ports in $\overline{mult}$, $(done, \overline{start}, \overline{p}, o)$ to *fac*.

2. Replace the state transition containing $v = u * v$ with an *implementation* of $\overline{mult}$, by merging the start state with the source state of the transition and the

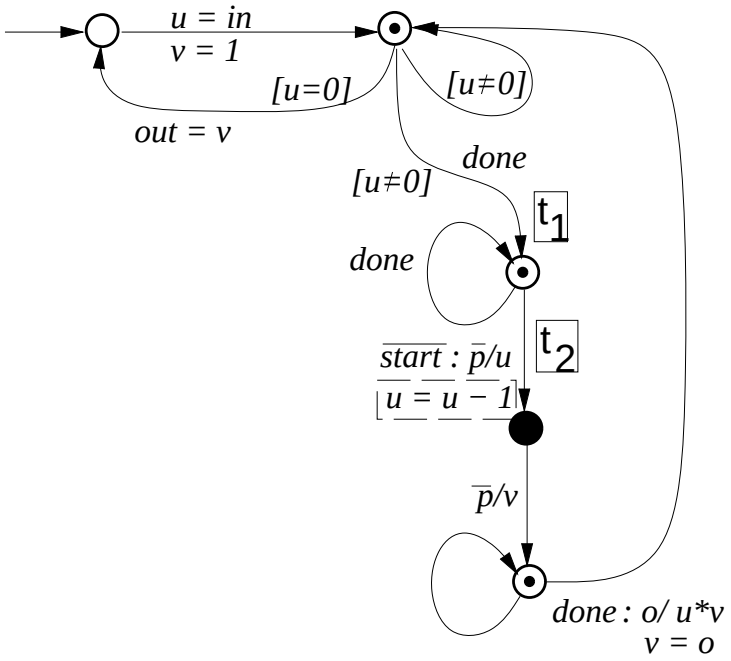

Figure 6.2: Factorial Machine with Multiplier Factored

final state with the target state of the transition (Figure 6.2).

3. The previous step does not take into account the other procedure $u = u-1$ in the replaced transition in the original *fac*. This procedure is scheduled in the earliest possible transition that does not go from a state to itself, such that the register value dependencies in the original *fac* are preserved. For simple expressions this is done by textual comparison. In general this involves verification of equivalence of logical and arithmetic expressions, and is a heuristic task [19].

Let û, v̂ be the values in registers u, v before the procedures $u = u-1$ and $v = u*v$ in the original *fac*. The values in the registers after the procedures in the original *fac* and by scheduling $u = u - 1$ with $\boxed{t_1}$ and $\boxed{t_2}$ in the modified *fac* (Figure 6.2) are shown in the table below. $\boxed{t_2}$ is the earliest transition where the procedure $u = u - 1$ gives the same values as the original *fac*.

	Register Values after Procedures	
	u	v
Original	$\hat{u} - 1$	$\hat{u} * \hat{v}$
$u = u - 1$ with t_1	$\hat{u} - 1$	$(\hat{u} - 1) * \hat{v}$
$u = u - 1$ with t_2	$\hat{u} - 1$	$\hat{u} * \hat{v}$

The modified *fac* can be turned into a machine with only the interface specification by hiding all the names in the original *fac* except the ones added in step 1 above. The modified *fac* and *mult* are subprocesses that together implement the original *fac*.

6.2 Scheme System Decomposition

The Scheme system is a special-purpose computer providing the symbolic processing primitives of the Scheme programming language. This example is part of a much larger exercise to derive hardware from denotational semantics of Scheme. The first step in this exercise is to derive an interpreter for Scheme and the next step involves decomposing the interpreter to a machine model and a compiler. The machine model operates on a heap object that includes an allocator, a garbage collector and a memory sub-system.

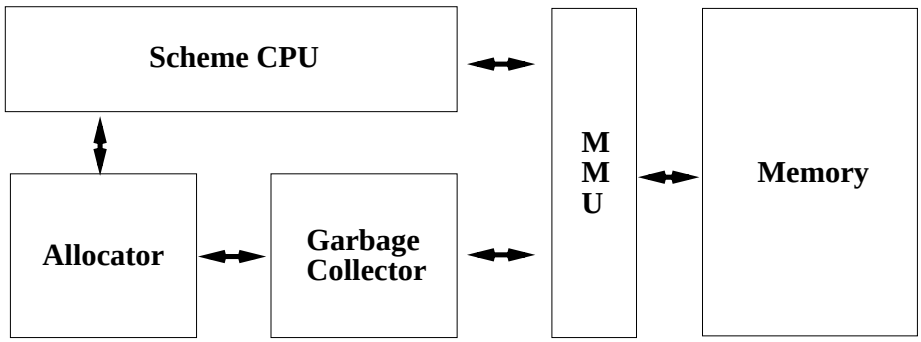

Figure 6.3: Scheme Machine System Organization

This example describes the decomposition of the machine model into a Scheme

CPU, an allocator, a garbage collector and dynamic RAM memory (Figure 6.3). Currently, each individual object has been derived and realized independently. The Scheme CPU has been derived to a realization in Actel FPGAs and PALs. The heap with two 4MB semi-spaces has been realized using Actel FPGAs, PALs, and dynamic RAMs. Our goal is to start the derivation at a higher level of specification and decompose it into interacting components.

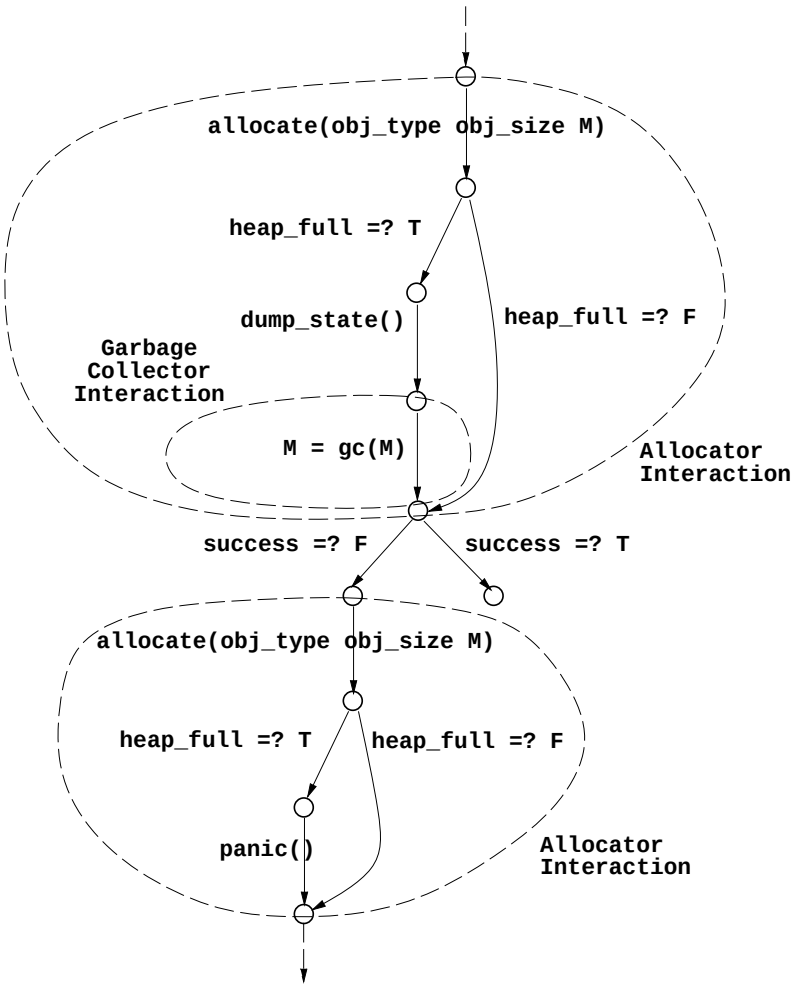

Figure 6.4: Fragment from Scheme Machine Specification

6.2.1 Allocator Decomposition

A fragment of the state diagram from the behavioral specification of the scheme machine is shown in Figure 6.4. Our first task will be to decompose the allocator from the scheme machine based on its interface specification. Since the garbage collection procedure is embedded within the allocator interaction (Figure 6.4), we will factor it into the allocator as an internal procedure.

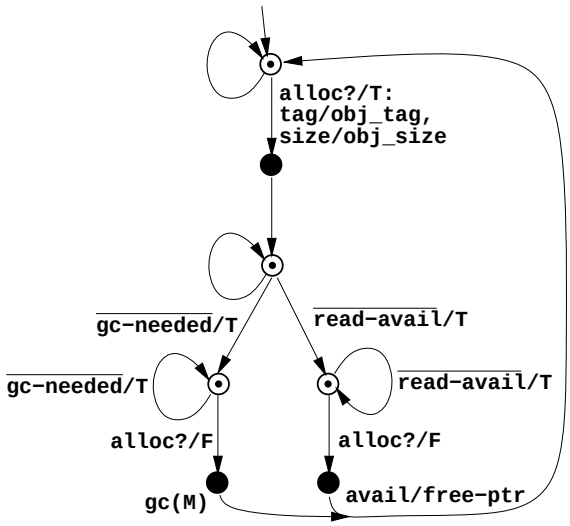

Figure 6.5: Allocator State Diagram

The ISL specification for the allocator is shown below:

$$\text{Allocator(alloc?, tag, size, avail, } \overline{\text{read-avail}}, \overline{\text{gc-needed}})(M) \triangleq$$
$$[\ \textit{await}\ \text{alloc?}/T\ :\ \text{tag/obj-tag}\ ,\ \text{size/obj-size}\ ;\ ;$$
$$((\overline{\text{read-avail}}/T\ \textit{until}\ \text{alloc?}/F\ ;\ \text{avail/free-ptr}\ ;\ )$$
$$[]\ \ (\overline{\text{gc-needed}}/T\ \textit{until}\ \text{alloc?}/F\ ;\ \{M = gc(M)\}\ ))\ ]^*$$

The state diagram for the allocator is shown in Figure 6.5. An implementation of an interaction path of the complement of the allocator can be embedded in the scheme

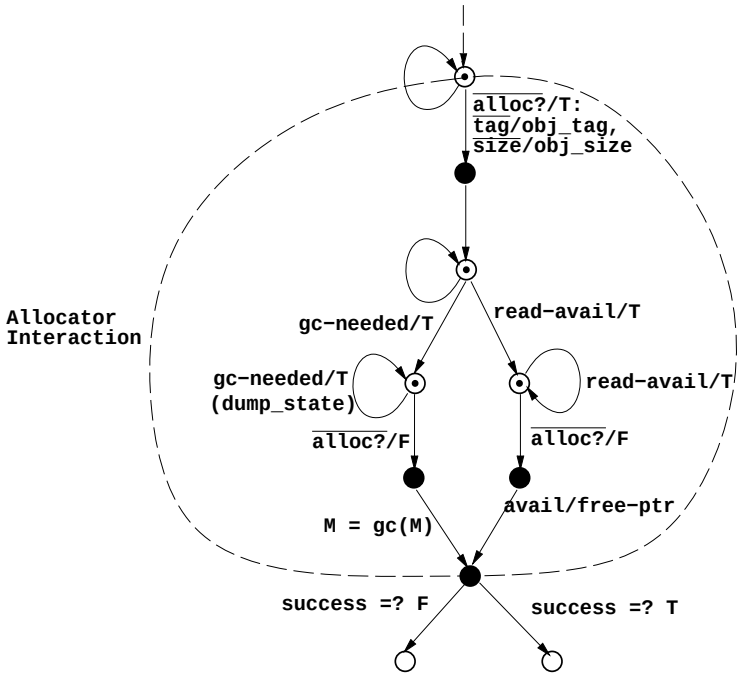

Figure 6.6: Embedding path implementation of $\overline{\text{Allocator}}$

machine, in place of the sequence of transitions for allocation interaction in the scheme machine specification (Figure 6.4). 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. The merge operation on states is defined in Section 4.2.2. The target state for the sequence of transitions for allocator interaction is also merged with the final state. Figure 6.6 shows the embedding of a path implementation of the allocator replacing one of the sequence of transitions for allocator interaction in Figure 6.4. 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 $\overline{\text{Allocator}}$, 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 $\overline{\text{Allocator}}$ (alloc?, tag, size, avail, $\overline{\text{read-avail}}$, $\overline{\text{gc-needed}}$) are added to the

scheme CPU.

6.2.2 Garbage Collector Decomposition

The next step in the derivation is to factor the garbage collector from the allocator. The path implementation of the garbage collector cycle (Figure 5.2) is then embedded into the allocator by replacing the sequence of transitions labeled $M = gc(M)$, as shown in Figure 6.7 (part of Scheme machine specification in Figure 6.4). 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 $\overline{\text{gc}}$ ($\overline{\text{collect}}$, gc-active) are added to the allocator.

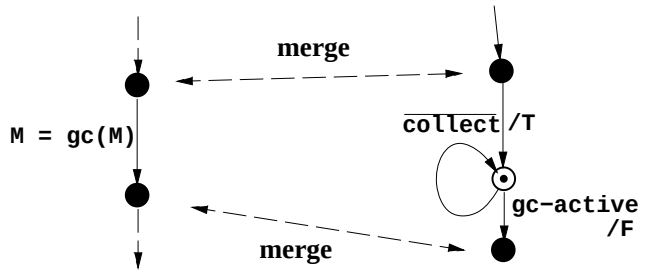

Figure 6.7: Embedding path implementation of $\overline{\text{gc}}$ in Allocator

We decomposed the scheme machine into a scheme CPU and an allocator, and then decomposed 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.

6.3 DDD-FM9001 Decomposition

A realization of the Nqthm FM9001 [37] specification, called DDD-FM9001 [6], was derived using the DDD [42] derivation system. DDD is a set of mechanized transformation tools to derive boolean descriptions from iterative functional specifications.

The tool set includes a transformation called system factorization that is used to factor parts of a system specification into functional objects with trivial interactions between them.

The derivation involved using system factorization to decompose the memory component. System factorization imposed restrictions on the design limiting the memory to a static RAM realization. We now look at this example in the context of sequential decomposition to realize a memory interface for dynamic RAM.

The DDD-FM9001 is a general purpose microprocessor realized in FPGAs, mechanically derived from Hunt's Nqthm FM9001 specification [37]. The FM9001 is a 32-bit microprocessor mechanically verified in the Nqthm theorem prover and implemented in LSI Logic's gate array technology. Details of the derivation of the DDD-FM9001 are reported in [6].

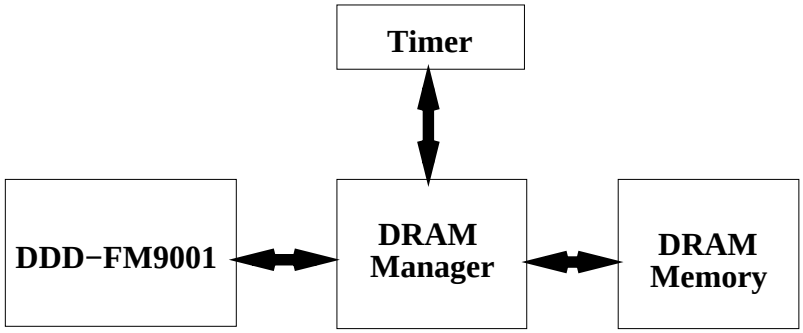

Figure 6.8: System Organization

Starting with specifications for DRAM and DDD-FM9001, let us derive the system organization shown in Figure 6.8.

The state machine denoted by the DRAM specification is shown in Figure 4.7. The read, write and refresh cycles are represented as three paths from the reset state to the final state. The environment of the DRAM ($\overline{\text{DRAM}}$), constructed using the complement operation, will be transformed into a path implementation that can interact with the DDD-FM9001 and a refresh timer.

6.3.1 Refresh Timer Derivation

The DRAM specification lacks the information that a refresh cycle must be performed within 4ms of the previous refresh cycle. For a clock speed of 10Mhz, allowing a read/write cycle time of 400ns and refresh cycle time of 300ns, we use a 3.3ms timer to start the refresh cycle. For slower clock speeds the refresh timer must be initialized appropriately. The timer is set at the end of each refresh cycle and holds the $\overline{\text{done}}$ signal until it is set again. The timer can be specified in ISL as :

$$\text{Timer}(\text{set}, \overline{\text{done}}) \triangleq (\ ; \overline{\text{done}} \ \textit{until} \ \text{set})^*$$

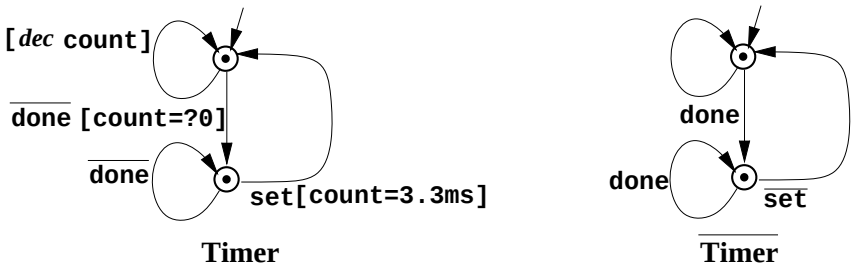

Figure 6.9: Timer and its complement

The state diagram for Timer (Figure 6.9) also shows its internal behavior within square parenthesis. The internal register count is loaded with a value equivalent to 3.3ms with set. It is decremented and tested in the next wait state. The internal behavior of the Timer is not considered in the complement operation on the Timer (Figure 6.9).

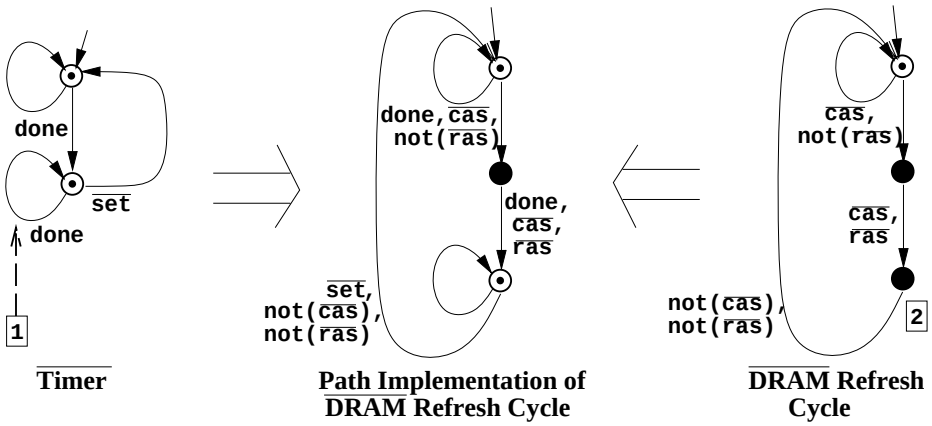

Figure 6.10: Transformation of $\overline{\text{Timer}}$

The next step in the derivation is to transform the complement of Timer ($\overline{\text{Timer}}$) into a path implementation of $\overline{\text{Timer}}$ and $\overline{\text{DRAM}}$ (refresh cycle). The wait transition labeled done (marked 1 in Figure 6.10) is unrolled into a transit state and a transition with the same label. This is a valid transformation because the outgoing transition from the wait state only has valid control outputs and no valid control inputs. The state marked 2 in Figure 6.10 is transformed into a wait state. This transformation is also valid because the outgoing transition from the state marked 2 has no valid control inputs. The labels in corresponding transitions of the two path implementations are then unified.

6.3.2 Derivation of Read and Write Cycles

In the next step of the derivation, the read and write operations on the abstract memory in DDD-FM9001 are decomposed into a sequential component using the ISL specifications shown in Figure 6.11. The complements of the `Read` and `Write` descriptions are then transformed into a path implementation of $\overline{\text{DRAM}}$.

$$\texttt{Read}(\text{dtack}, \overline{\text{strobe}}, \overline{\text{RW}}, \overline{\text{ADDR}}, \overline{\text{DOUT}}) \triangleq$$
$$[;\ \overline{\text{strobe}}, \overline{\text{RW}} : \overline{\text{ADDR}}/V_{\text{addr}}\ \textit{until}\ \text{dtack} : \text{DIN}/V_{\text{read}}]$$
$$\texttt{Write}(\text{dtack}, \overline{\text{strobe}}, \overline{\text{RW}}, \overline{\text{ADDR}}, \text{DIN}) \triangleq\ [;\ \overline{\text{strobe}},$$
$$\text{not}(\overline{\text{RW}}) : \overline{\text{ADDR}}/V_{\text{addr}}, \overline{\text{DOUT}}/V_{\text{write}}\ \textit{until}\ \text{dtack}]$$

Figure 6.11: DDD-FM9001 Read and Write Interface Specifications

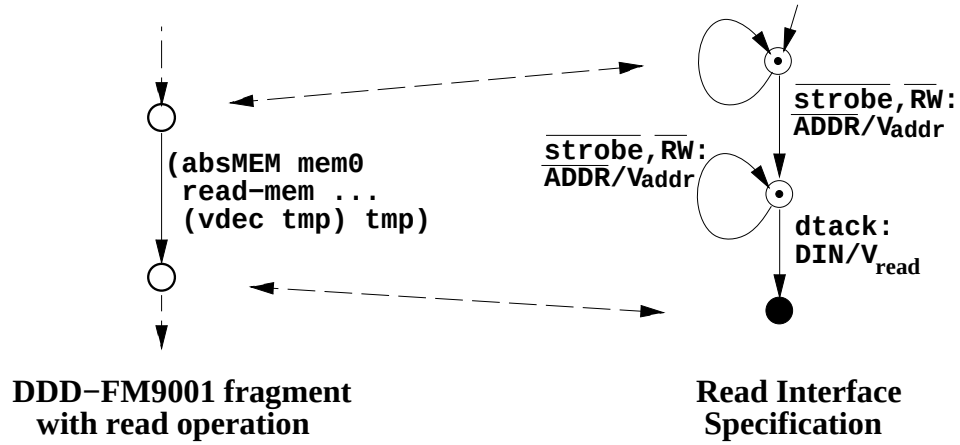

Figure 6.12: DDD-FM9001 Read Interface

As shown in Figure 6.12, we incorporate the interface for `Read` into a lower level descripton of the DDD-FM9001 by replacing each transition containing the read-mem operation with `Read`. The ports in `Read` ($\overline{\text{strobe}}$, $\overline{\text{RW}}$, $\overline{\text{ADDR}}$, $\overline{\text{DOUT}}$, DIN, dtack) are added to the DDD-FM9001 description. Similarly, the write-mem operations in DDD-FM9001 are replaced by instances of `Write`.

Figure 6.13 shows the transformation of $\overline{\texttt{Read}}$ to a path implementation of $\overline{\text{DRAM}}$ (read cycle). The wait loop in the state marked $\boxed{3}$ in Figure 6.13 is unrolled twice. The state marked $\boxed{4}$ in Figure 6.13 is transformed into a wait state. The labels of the resulting state diagram are then unified. Similarly, $\overline{\texttt{Write}}$ is transformed to a path implementation of $\overline{\text{DRAM}}$ (write cycle).

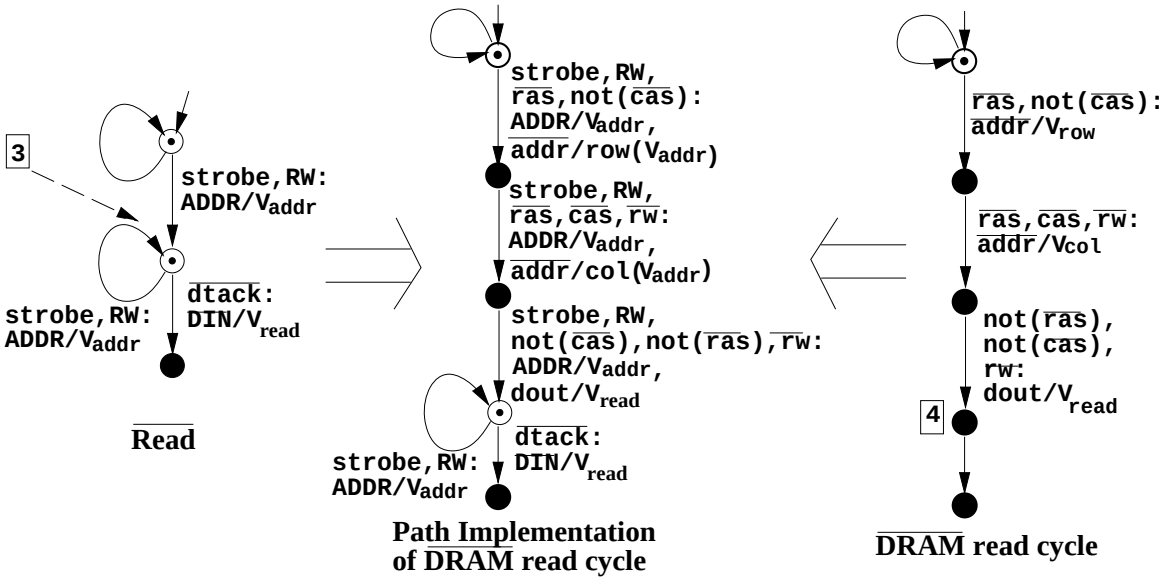

Figure 6.13: Transformation of Read

Chapter 7

Conclusions and Future Directions

This dissertation addressed one of the key problems in the design of large systems with distributed control and non-trivial interactions between components. Design of a digital system involves both top-down and bottom-up reasoning. Our emphasis here has been on goal directed derivation to promote it as an alternative to strictly top-down or bottom-up techniques. We described a method for the decomposition of system specifications into interacting components with the interface between them derived from the protocol specifications of the components. This is an extension of the *system factorization* [40] transformation which is used to encapsulate abstract values and operations as simple sequential processes with trivial protocols. Sequential decomposition includes support for arbitrary interaction protocols between components in a system.

The decomposition method described here does not state anything about functional correctness of a decomposed sequential component, the ability of the component to perform the factored operation. All the examples in the dissertation assume that the decomposed components are functionally equivalent to the procedures or operations they replace.

Consider the factorial machine decomposition in Section 6.1. We assume that the sequential multiplier performs the multiplication function because a term matching the multiplication operation on the input data terms is assigned to an output port in the multiplier specification. In the Scheme machine example (Section 6.2),

the garbage collector specification abstracts its functional behavior into the *garbage-collect(M)* function. We assume this function to be equivalent to the garbage collection procedure in the Scheme system specification. Similarly, in the DDD-FM9001 decomposition example (Section 6.3), the memory read and write operations in the DDD-FM9001 are abstracted into equivalent functions in the DRAM specification.

Let us now suggest a sufficient criteria for functional correctness of a decomposed component. Given that the original specification has a transition label that assigns a term for the result of the procedure to be decomposed to a port or register. The interface specification of the factored component must assign the same or an equivalent term to an output port. Also, the residual machine must make an assignment from the input port, corresponding to the output port in the decomposed component, to the original destination register or port. The transition path in the component with the matching term forms the basis for choosing the *path implementation* of its complement that is grafted into the original specification. The criteria described above can not deal with factoring sub-terms from specifications. Zhu describes an algebra in [82] can be used in a more general treatment of functional correctness.

The decomposition technique described here was developed for designing systems with globally synchronous communications and a common synchronizing reference. Many of the concepts presented here are similar to asynchronous self-timed design methods. Asynchronous systems require a synchronizing control signal on every interaction. Our protocol specification language ISL has asynchronous communication primitives (1-way and 2-way communication) and a synchronous communication primitive (lock-step communication). It can be adapted for use in asynchronous designs by eliminating the synchronous primitive.

Compared to system synthesis methods, the derivation method described here 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 encapsulating a procedure in a sequential machine with a specified interaction protocol. With each transformation step the designer extracts a sequential component from the system and incorporates an implementation of the complement of the interaction protocol description of the sequential component into the system. All the components in the system can be then be derived into hardware.

In this dissertation we described a language for interface specification and a finite state machine based semantic model. This language can be used to describe the interaction among components in a synchronous hardware system. A definition in this language describes the input/output behavior of a machine. A set of operations and relations on machines were also defined. The complement operation is used to construct the most general environment of a machine. An implementation of the complement of this machine is incorporated in the original machine. Successive decomposition steps result in an implementation of a network of machines that implement the high-level specification.

We described an isomorphism relation over machines and also developed the implementation and path implementation relations. We were also introduced to the notion of safe interaction between machines. The following results were proved:

1. A machine composed with its complement results is a closed machine that is isomorphic to the original machine. The isomorphism of the composed machine shows that the temporal characteristics of a machine are preserved in composition with its complement.

   This result shows that a component in a system is capable of successfully completing all its interactions if its environment behaves as its complement. A system that is composed of a machine and its complement means that all outputs generated by the machine are accepted by its complement and all inputs required by the machine are generated by its complement. Such a system does not have any requirements from outside the system and is called a closed system.

2. A machine composed with a path implementation of its complement results in a safe interaction. The safety of the composed machine implies that there is some interaction sequence between the interacting machines that completes a protocol.

   The intuition behind this result is that a machine can have a useful interaction with its environment and safely complete a protocol from start to end if it gets the right values when it expects them and outputs the right values when its environment expects them.

There are four facets to system design, control, architecture, data representation and protocol. The DDD tool set has many transformations for the control, architecture and data representation facets, but can only generate an interface for a component with a trivial protocol. The decomposition method developed here advances the capabilities of the tool set by generalizing the system factorization transformation to handle arbitrary protocols.

The scheme machine provided a substantial example for the decomposition method described in this dissertation. Sequential decomposition was used to decompose a scheme system into a scheme CPU, an allocator, and a garbage collector with non-trivial interactions between the components. We also looked at the steps involved in the derivation of a DRAM based memory system using sequential decomposition for the DDD-FM9001 processor. The specifications of the DRAM memory protocol, and the interactions of the timer and the read and write operations by the DDD-FM9001 were used in the derivation.

Currently, there is a lot on interest in the research community in looking at the problems in distributed system design. Many researchers have looked at verification of protocols in a system model by testing its invariant properties. Others have attempted to automatically generate interface logic to go between interacting components in a system. Each approach has its limitations, either in the kinds of properties that can be verified, or in the kinds of protocols that are amenable to automatic interface

logic generation. The method described here supports arbitrary protocols but is less automatic. A designer can use sequential decomposition to partition a system description and use other transformations to derive an implementation.

7.1 Limitations of the Approach

One of the limitations of ISL is that there is no mechanism to quantify time in the language and all timing constraints must be specified as an ordering of events based on the system synchronization clock. For example, the refresh timer in the DDD-FM9001 example (Figure 6.9) could not be specified to go off every 3.3ms. Instead it was specified as a value to be loaded in a counter with respect to a clock speed.

Another limitation is that the interface stubs can not be generated automatically and must be transformed from the complement of the specification on the other side of the interaction into a path implementation. This is probably due to the generality of the protocol specifications. As we have seen in the examples in Chapter 6, all the transformations were goal directed and were performed with knowledge of the resulting path implementation.

7.2 Behavior Tables

System level design can be described as transformations on four orthogonal facets of system specifications, control, architecture, protocol and data representation. Behavior tables are an extension of register-transfer tables and provide a unified basis for system representation for reasoning about all facets of system design [66]. Starting from a specification with symbolic data values, transformations can be used by a designer to direct the design towards an implementation of interacting behavior tables with boolean data representation and appropriate control and datapath abstractions.

The decomposition of system specifications and reasoning about protocol aspects of system design can be addressed within the framework of behavior tables. Behav-

ior tables are a representation of the finite state machine model that is used as the semantic model for ISL. Thus, the interaction of a component with its environment specified in ISL can be easily translated into a behavior table. Also, all the implementation relations and transformations described in this dissertation can be applied to behavior tables.

7.3 Future Directions

In the future, I would like to explore other problem areas to which this methodology could be applied, especially in hardware-software co-design problems. Since ISL models system behavior as values over ports, it can also be used to model software processes. There is no mechanism to express data abstractions in ISL and that might be a problem in software specification. The decomposition method should be useful for both hardware and software components.

The use of this technique to decompose asynchronous and self-timed circuits should also be explored. The interface specification language will need modifications to support a asynchronous model instead of the current synchronous model. Also, the semantic model of the language will have to be modified for asynchronous system design. The general methodology of decomposition will probably be applicable for asynchronous designs.

The system decomposition method described here can be of use if it is included in the DDD design derivation tool set [42]. An integrated design environment based on behavior tables with transformations for different design facets must be implemented for a designer to make use of all the ideas described in this dissertation. A system design framework should model the four orthogonal facets of system design, control, architecture, protocol and data representation. Also, an ideal design environment should allow a designer to explore the design space of different implementations of a specification using transformations. The design methodology presented here contributes some ideas towards exploring the protocol facet of a design.

Bibliography

[1] T. K. Lee D. Borkovic A. J. Martin, S. M. Burns and P. J. Hazewindus. The design of an asynchronous microprocessor. In *Proc. Decennetial Caltech Conference on VLSI.* CS Dept, California Institute of Technology, Pasadena, CA, MIT Press, March 1989.

[2] Venkatesh Akella. hopCP: A new paradigm for VLSI programming. VLSI Systems Research Group, Note VLSI-89-002, University of Utah, Department of Computer Science, August 1989.

[3] W. R. Bevier and W. D. Young. The proof of correctness of a fault-tolerant circuit design. In *Proceedings of International Conference on Dependable Computing for Critical Applications*, pages 107–114, February 1991.

[4] G. Borriello. Specification and synthesis of interface logic. *High-Level VLSI Synthesis*, pages 153–176, 1991.

[5] Bhaskar Bose. DDD - A Transformation system for Digital Design Derivation. Technical Report 331, Department of Computer Science, Indiana University, May 1991.

[6] Bhaskar Bose and Steven D. Johnson. DDD-FM9001: Derivation of a verified microprocessor. an exercise in integrating verification with formal derivation. In *Proceedings of IFIP Conference on Correct Hardware Design and Verification Methods.* Springer, 1993.

[7] Bhaskar Bose, Steven D. Johnson, and Shyam Pullela. Integrating boolean verification with formal derivation. In D. Agnew, L. Claesen, and R. Camposano,

editors, *Proceedings of IFIP Conference on Hardware Description Languages and their Applications*, pages 127–134. Elsevier, April 1993.

[8] Bishop Brock and Warren A. Hunt. The FM9001 proof script. Technical report, Computational Logic Incorporated, 1993.

[9] R.E. Bryant. A switch level model and simulator for MOS digital circuits. *IEEE Transactions on Computers*, C-33(2):160–177, February 1984. COSMOS paper.

[10] S.M. Burns. Automated compilation of concurrent programs into self-timed circuits. Technical Report Caltech-CS-TR-88-2, California Institute of Technology, Computer Sc Dept, Caltech, Pasadena, CA, December 1987.

[11] Tam Anh Chu. Synthesis of self timed VLSI circuits from graph theoritic specifications. In *Intl. Workshop on Petri Nets and Performance Models*, August 1987.

[12] Ed Clarke, D. Dill, J. Burch, K. L. McMillan, and L. J. Hwang. Symbolic model checking: 10**20 states and beyond. In *International Workshop on Formal Methods in VLSI Design*. ACM-SIGDA, January 1991.

[13] Edmund M. Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David E. Long, and Linda A. Ness. Verification of the futurebus+ cache coherence protocol. In *Proceedings of IFIP Conference on Hardware Description Languages and their Applications (CHDL)*, 1993.

[14] E.M. Clarke, D.E. Long, and K.L. McMillan. A language for compositional specification and verification of finite state hardware controllers. *Proceedings of the IEEE*, 79, September 1991.

[15] Rance Cleaveland and Matthew Hennessy. Testing equivalence as a bisimulation equivalence. In Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, pages 11–23. Springer Verlag, 1990. LNCS 407.

[16] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench. In Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, pages 24–37. Springer Verlag, 1990. LNCS 407.

[17] Bruce S. Davie. *A Formal, Hierarchical Design and Validation Methodology for VLSI.* PhD thesis, University of Edinburgh, 1988.

[18] Bruce S. Davie and George J. Milne. Contextual constraints for design and verification. In Birtwistle and Subramanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 257–265. Kluwer, 1988.

[19] S. Devadas and K. Keutzer. An Automata-Theoretic Approach to Behavioral Equivalence. In *Proceedings of the International Conference on Computer-Aided Design*, pages 30–33, November 1990.

[20] Srinivas Devadas and A. Richard Newton. Decomposition and factorization of sequential finite state machines. *Transactions on Computer-Aided Design 1989*, 8(11):1206–1217, November 1989.

[21] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* MIT Press, 1988.

[22] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *Proceedings of International Conference on Computer Design: VLSI in Computers & Processors*, pages 522–525. IEEE, 1992.

[23] David L. Dill, Alan J. Hu, and Howard Wong-Toi. Checking for language inclusion using simulation preorders. In Larsen and Skou, editors, *Proceedings of Computer Aided Verification*, pages 255–265. Springer, July 1991. LNCS 575.

[24] David L. Dill, Steven M. Novick, and Robert F. Sproull. Automatic verification of speed-independent circuits with petri net specification. In *IEEE International*

Conference on Computer Design, pages 212–216. Stanford Univ., IEEE Press, 1989.

[25] Doron Drusinsky and David Harel. Using statecharts for hardware description and synthesis. *Transactions on CAD*, 8(7):798–807, July 1989.

[26] J. C. Ebergen. Arbiters: An exercise in specifying and decomposing asynchronously communicating components. CS Dept, University of Waterloo.

[27] Daniel D. Gajski. *High-Level VLSI Synthesis*, chapter Essential Issues and Possible Solutions in High-Level Synthesis, pages 1–26. Kluwer, 1991.

[28] Ganesh Gopalakrishnan and Prabhat Jain. Some recent asynchronous system design methodologies. Technical Report UU-CS-TR-90-016, Dept. of Computer Sc, University of Utah, Salt Lake City, Utah 84112, October 1990.

[29] Ganesh C. Gopalakrishnan. Specification and verification of pipelined hardware in HOP. In J.A.Darringer and F.J.Rammig, editors, *Computer Hardware Description Languages*, pages 117–131. ELSEVIER, 1989.

[30] Ganesh C. Gopalakrishnan, Richard M. Fujimoto, Venkatesh Akella, and Narayana S. Mani. HOP: A process model for synchronous hardware; semantics and experiments in process composition. *Integration, the VLSI journal*, 8:209–247, 1989.

[31] Ganesh C. Gopalakrishnan, Narayana S. Mani, and Venkatesh Akella. Parallel composition of lockstep synchronous processes for hardware validation: divide-and-conquer composition. In A. Pneuli, J. Sifakis, and E. Clarke, editors, *Automatic Verification Methods for Finite State Systems*, Springer Lecture Notes in Computer Science. Springer, Berlin, 1990. Proceedings of the 1989 C-cube workshop, Grenoble, in press.

[32] Matthew Hennessy. *Algebraic Theory of Processes.* MIT Press, Cambridge, MA, 1988.

[33] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[34] Gerard J. Holzmann. Tracing protocols. In Yemini, editor, *Current Advances in Distributed Computing and Communications*, pages 189–207. Computer Science Press Inc, 1987.

[35] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, 1979.

[36] Warren A. Hunt. Microprocessor design verification. *Journal of Automated Reasoning*, 5:429–460, 1989.

[37] Warren A. Hunt. A formal HDL and its use in the FM9001 verification. In C.A.R. Hoare and M.J. Gordon, editors, *Mechanized Reasoning in Hardware Design.* Prentice-Hall, 1992.

[38] Steven D. Johnson. Applicative programming and digital design. In *Proceedings of 11th Annual SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 218–227, 1984.

[39] Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations.* MIT Press, Cambridge, 1984. ACM Distinguished Dissertation 1984.

[40] Steven D. Johnson. Manipulating logical organization with system factorizations. In Leeser and Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, volume 408 of *LNCS*, pages 260–281. Springer, July 1989. Proceedings of Mathematical Sciences Institute Workshop, Cornell University, 1989.

[41] Steven D. Johnson and B. Bose. A system of digital design derivation. Technical Report 289, Indiana University, Computer Science Department, Indiana University, August 1989.

[42] Steven D. Johnson and Bhaskar Bose. A system for mechanized digital design derivation. In Subramanyam, editor, *Proceedings of ACM International Workshop on Formal Methods in VLSI Design*, January 1991.

[43] Steven D. Johnson, R.M. Wehrmeister, and B. Bose. On the interplay of synthesis and verification: Experiments with the FM8501 processor description. In Claesen, editor, *Applied Formal Methods for Correct VLSI Design*, pages 385–404. Elsevier, 1989. IMEC 1989.

[44] Jeffrey J. Joyce. Formal verification and implementation of a microprocessor. In C. Birtwhistle and P.A Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*. Academic Press, 1988.

[45] L. Jozwiak and J. Kolsteren. An efficient method for the sequential general decomposition of sequential machines. *Microprocessing and Microprogramming*, 31:657–664, 1991.

[46] Lech Jozwiak. Simultaneous decompositions of sequential machines. *Microprocessing and Microprogramming*, 30:305–312, 1990.

[47] David Ku and Giovanni De Micheli. Relative scheduling under timing constraints. In *Proceedings of ACM/IEEE Design Automation Conference*, June 1990.

[48] Andreas Kuehlmann and Reinaldo A. Bergamaschi. High-level state machine specification and synthesis. In *Proceedings of International Conference on Computer Design: VLSI in Computers & Processors*, pages 536–539. IEEE, 1992.

[49] R. P. Kurshan. Analysis of discrete event simulation. In Bakker, Roever, and Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, pages 414–453. Springer-Verlag, July 1989. LNCS 430.

[50] I.S. Levin. A hierarchical model of the interaction of micropragrammed automata. *Avtomatika i Vychislitelnaya Technika*, 21(3):77–83, 1987. Translation from Russian by Allerton Press.

[51] A. J. Martin. A synthesis method for self-timed VLSI circuits. In *ICCD87: 1987 IEEE International Conference on Computer Design*, pages 224–229. CS Dept, California Institute of Technology, Pasadena, CA, IEEE Computer Society Press, 1987.

[52] Kayhan Küçükçakar and Alice C. Parker. CHOP: A constraint-driven system-level partitioner. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 514–519, 1991.

[53] Kenneth L. McMillan and David L. Dill. Algorithms for interface timing verification. In *Proceedings of IEEE International Conference on Computer Design*, pages 48–51. IEEE Computer Society, November 1992.

[54] George J. Milne. CIRCAL: A calculus for circuit description. *Integration*, 1:121–160, 1983.

[55] George J. Milne. Design for verifiability. In Leeser and Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, pages 1–13. Springer, July 1989. LNCS 408.

[56] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[57] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[58] Robin Milner. Elements of interaction. *Communications of the ACM*, 36(1):78–89, January 1993. Turing award lecture.

[59] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes - i and ii. *Information and Computation*, 100(1):1–40,41–77, September 1992.

[60] Paul Miner. Verification of fault-tolerant clock synchronization systems. Technical Report 3349, National Aeronautics and Space Administration, Hampton, VA, November 1993.

[61] David E. Muller and W. S. Bartky. A theory of asynchronous circuits. *The Annals of the Computation Laboratory of Harvard University*, 29:204–243, April 1959.

[62] J. A. Nestor and D. Thomas. Behavioral synthesis with interfaces. In *Proceedings of ICCAD*, November 1986.

[63] Xavier Nicollin and Joseph Sifakis. An overview and synthesis on timed process algebras. In Larsen and Skou, editors, *Proceedings of International Conference on Computer Aided Verification*, pages 376–398. Springer, July 1991. LNCS 575.

[64] Steven M. Nowick, Kenneth Y. Yun, and David L. Dill. Practical asynchronous controller design. In *Proceedings of International Conference on Computer Design*, pages 341–345. IEEE, 1992.

[65] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Denmark, 1981.

[66] Kamlesh Rath, M. Esen Tuna, and Steven D. Johnson. Behavior tables: A basis for system representation and transformational system synthesis. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*. IEEE, November 1993.

[67] Klaus Schneider, Ramayya Kumar, and Thomas Kropf. Hardware verification using first order BDDs. In *Proceedings of IFIP Conference on Hardware Description Languages and their Applications (CHDL)*, 1993.

[68] Carl-Johan Seger and Jeffrey J. Joyce. A two-level formal verification methodology using HOL and COSMOS. In *Proceedings of Computer Aided Verification Conference*, pages 299–309. Springer, July 1991. LNCS 575.

[69] Robin Sharp and Ole Rasmussen. Rewriting and constraints in t-ruby. In Milne and Pierre, editors, *IFIP Conference on Correct Hardware Design and Verification Methods*, pages 226–241. Springer, May 1993. LNCS 683.

[70] Mary Sheeran. muFP, An algebraic VLSI design language. In *ACM Symposium on LISP and Functional Programming*, pages 104–112, 1984.

[71] Mary Sheeran. Retiming and slowdown in ruby. In Milne, editor, *The fusion of hardware design and verification*, pages 285–308. IFIP, North-Holland, 1988.

[72] Andrés R. Takach and Wayne Wolf. Behavior FSMs for high-level synthesis and verification. Technical Report CE-W91-13, Dept. of Electrical Engineering, Princeton University, July 1991.

[73] Texas Instruments. *MOS Memory Data Book*, 1989.

[74] Frank Vahid and Daniel D. Gajski. Specification partitioning for system design. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pages 219–224, 1992.

[75] Robert A. Walker and Donald E. Thomas. Behavioral transformation for algorithmic level IC design. *IEEE Transactions on Computer-Aided Design*, 8(10):1115–1128, 1989.

[76] Wayne Wolf, Andrés Takach, and Tien-Chien Lee. Architectural optimization methods for control-dominated machines. *High-Level VLSI Synthesis*, pages 231–254, 1991.

[77] Maya K. Yajnik and Maciej J. Ciesielski. Finite state machine decomposition using multi-way partitioning. In *Proceedings of International Conference on Computer Design: VLSI in Computers & Processors*, pages 320–323. IEEE, 1992.

[78] Zheng Zhu. *Structured Hardware Design Transformations*. PhD thesis, Computer Science Department, Indiana University, USA, 1992.

[79] Zheng Zhu. Automatic synthesis of sequential synchronizations. In D. Agnew, L. Claesen, and R. Camposano, editors, *Proceedings of IFIP Conference on Hardware Description Languages and their Applications*, pages 285–301. Elsevier, April 1993.

[80] Zheng Zhu and Steven D. Johnson. An example of digital design transformation in an algebraic framework. In Subramanyam, editor, *Proceedings of ACM International Workshop on Formal Methods in VLSI Design*, January 1991.

[81] Zheng Zhu and Steven D. Johnson. Automatic synthesis of sequential synchronizations. In D. Agnew, L. Claesen, and R. Camposano, editors, *Proceedings of IFIP Conference on Hardware Description Languages and their Applications*, pages 285–301. Elsevier, April 1993. Also published as Technical Report No. 373, Dept. of Computer Science, Indiana University.

[82] Zheng Zhu and Steven D. Johnson. Capturing synchronization specifications for sequential compositions. In *Proceedings of the 1994 IEEE International Conference on Computer Design (ICCD 94)*, pages 117–121. IEEE, October 1994.

Appendix

Proof for Theorem 5.1.1

Each net of $\mathcal{N}_0$ is obtained by connecting an input (alternatively output) port c of M with an output (alternatively input) port gensym(c) of $\overline{M}$. From the definition of composition each net in $\mathcal{N}_0$ is a port in M_0. Therefore M_0 consists of only output ports and is closed.

The composed machine $M_0 = (M \parallel \overline{M})_{\mathcal{N}_0}$ has the following structure:

$M_0 = \langle S_0, T_0, r_0, f_0, P_0, V_0 \rangle$ where,

V_0	$=$	V	M and $\overline{M}$ have the same value sets V
r_0	$=$	$\langle r, r \rangle$	from the definition of composition
P_0	$=$	$CI_0 \cup CO_0 \cup DI_0 \cup DO_0$	
CI_0	$=$	$\{\}$	from definition of composition and $\mathcal{N}_0$
CO_0	$=$	$\{\langle c, \text{gensym}(c) \rangle \mid c \in CI \cup CO\}$	
DI_0	$=$	$\{\}$	
DO_0	$=$	$\{\langle d, \text{gensym}(d) \rangle \mid d \in DI \cup DO\}$	

We prove the following two conditions which together imply that M and M_0 are isomorphic:

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

1. $\langle s_i, s_j \rangle \in S_0 \iff s_i = s_j$

2. $\langle s_i, s_j \rangle \xrightarrow{\mathcal{L}_0} \langle s_i', s_j' \rangle \in T_0$

   $\iff$

   $\langle s_i, s_j \rangle \in S_0 \ \wedge \ s_i = s_j \ \wedge \ s_i' = s_j' \ \wedge \ s_i \xrightarrow{\mathcal{L}_i} s_i' \ \wedge \ s_j \xrightarrow{\mathcal{L}_j} s_j' \ \wedge$

   $\mathcal{L}_j = \text{Rename}(\mathcal{L}_i) \ \wedge \ \mathcal{L}_0([p]) = \mathcal{L}_{ij}(p)$ if $p \in care_{\mathcal{L}_i} \cup care_{\mathcal{L}_j}$, # otherwise.

We will prove the above conditions by induction on the length of the derivation of states and transitions in the construction of M_0.

Base Case : $\langle r, r \rangle \in S_0$, 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

$\langle s_i, s_j \rangle \in S_0 \ \wedge \ s_i = s_j \ \wedge \ s_i' = s_j' \ \wedge \ s_i \xrightarrow{\mathcal{L}_i} s_i' \in T \ \wedge \ s_j \xrightarrow{\mathcal{L}_j} s_j' \in \overline{T} \ \wedge$

$\mathcal{L}_j = \text{Rename}(\mathcal{L}_i) \ \wedge \ \mathcal{L}_0([p]) = \mathcal{L}_{ij}(p)$ if $p \in care_{\mathcal{L}_i} \cup care_{\mathcal{L}_j}$, # otherwise.

By choosing every port $p_i \in care_{\mathcal{L}_i}$, $\quad care_{\mathcal{L}_i} \cup care_{\mathcal{L}_j} \subseteq \bigcup_{i=1 \text{ to } k} [p_i]$

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

$\mathcal{L}_{ij}(p) = \mathcal{L}_{ij}(p_i) = \mathcal{L}_{ij}(p_j)$

$\Rightarrow \forall p, p' \in [p_i] \,.\, \mathcal{L}_{ij}(p) \equiv \mathcal{L}_{ij}(p')$

$\Rightarrow \langle s_i, s_j \rangle \xrightarrow{\mathcal{L}_0} \langle s_i', s_j' \rangle \in T_0$ and $\langle s_i', s_j' \rangle \in S_0$ for every derivation of length $n+1$.

To show the converse, assume that $\langle s_i, s_j \rangle \xrightarrow{\mathcal{L}_0} \langle s_i', s_j' \rangle \in T_0$ is derived in $n+1$ steps. From the antecedents of the rule for composition

$s_i \xrightarrow{\mathcal{L}_i} s_i' \in T$ and $s_j \xrightarrow{\mathcal{L}_j} s_j' \in \overline{T}$ and $\langle s_i, s_j \rangle \in S_0$

Since $\langle s_i, s_j \rangle \in S_0$ would have been derived in n steps, by induction hypothesis $s_i = s_j$.

$\mathcal{L}_0([p]) = \mathcal{L}_{ij}(p)$ if $p \in care_{\mathcal{L}_i} \cup care_{\mathcal{L}_j}$, # otherwise.

$\forall p, p' \in [p_j].\mathcal{L}_{ij}(p) \equiv \mathcal{L}_{ij}(p')$ and since $\mathcal{N}_0$ guarantees that $[p_j] = \{p_j, \text{gensym}(p_j)\}$.

$\mathcal{L}_{ij}(p_j) \equiv \mathcal{L}_{ij}(\text{gensym}(p_j))$

We have $s_i \xrightarrow{\mathcal{L}_i} s_i' \in T$, $s_j \xrightarrow{\mathcal{L}_j} s_j' \in \overline{T}$, $\mathcal{L}_j = \text{Rename}(\mathcal{L}_i)$, and $s_i = s_j$. Since M and $\overline{M}$ are deterministic and minimal, $s_j \xrightarrow{\mathcal{L}_s} s_j' \in T$; hence $s_i' = s_j'$.

□

Proof for Lemma 5.5.1

Choosing every port $p_i \in care_{\mathcal{L}_i} \,.\, \exists p_j \in care_{\mathcal{L}_j} \wedge p_j = \text{gensym}(p_i)$

$$care_{\mathcal{L}_i} \cup care_{\mathcal{L}_j} \subseteq \bigcup_{i=1 \text{ to } k} [p_i]$$

Each net $[p_i]$ consists of two ports p_i and p_j. From the definition of *complementation* and *inclusion*, $\forall p_i \in care_{\mathcal{L}_i} \,.\, \mathcal{L}_i(p_i) = \overline{\mathcal{L}_i}(p_i) = \mathcal{L}_j(p_j)$.

$\mathcal{L}_{ij}(p) = \mathcal{L}_{ij}(p_i) = \mathcal{L}_{ij}(p_j) \;\Rightarrow\; \forall p, p' \in [p_i] \,.\, \mathcal{L}_{ij}(p) \equiv \mathcal{L}_{ij}(p')$

Therefore, $\mathcal{L}_i, \mathcal{L}_j$ are composable. □

Proof for Theorem 5.5.1

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

$\langle s_1, s_2 \rangle$ is reachable $\iff$

$$\overline{s_1} \sqsubset_{\text{p}} s_2 \;\wedge\; \forall s_1 \xrightarrow{\mathcal{L}_1} s_1', s_2 \xrightarrow{\mathcal{L}_2} s_2' \,.\, (\overline{\mathcal{L}_1} \preceq \mathcal{L}_2 \;\Rightarrow\; \overline{s_1'} \sqsubset_{\text{p}} s_2' \;\wedge\; \langle s_1, s_2 \rangle \rightarrow \langle s_1', s_2' \rangle \in T)$$

$\Rightarrow$:

Base Case :

$\overline{r_1} \sqsubset_{\text{p}} r_2$, from the definition of *complement* and *path implementation*.

From the definition of *path simulation*,

$\overline{r_1} \sqsubset_{\text{p}} r_2 \Rightarrow \forall \overline{r_1} \xrightarrow{\overline{\mathcal{L}_1}} \overline{s_1'}, r_2 \xrightarrow{\mathcal{L}_2} s_2' \,.\, \overline{\mathcal{L}_1} \preceq \mathcal{L}_2 \Rightarrow \overline{s_1'} \sqsubset_{\text{p}} s_2'$

Since, for every transition $\overline{r_1} \xrightarrow{\overline{\mathcal{L}_1}} \overline{s_1'} \in \overline{M_1}$, there is a transition $r_1 \xrightarrow{\mathcal{L}_1} s_1' \in M_1$,

$\forall r_1 \xrightarrow{\mathcal{L}_1} s_1', r_2 \xrightarrow{\mathcal{L}_2} s_2' \,.\, \overline{\mathcal{L}_1} \preceq \mathcal{L}_2 \Rightarrow \overline{s_1'} \sqsubset_{\text{p}} s_2'$

Using lemma 5.5.1, $\overline{\mathcal{L}_1} \preceq \mathcal{L}_2 \Rightarrow \langle r_1, r_2 \rangle \rightarrow \langle s_1', s_2' \rangle \in T$

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 $\langle s_{1_0}, s_{2_0} \rangle \xrightarrow{\mathcal{L}_0} \langle s_1, s_2 \rangle$,

for some $\langle s_{1_0}, s_{2_0} \rangle$, such that,

$\langle s_{1_0}, s_{2_0} \rangle \in M$ is reachable

$s_{1_0} \stackrel{\mathcal{L}_{1_0}}{\longrightarrow} s_1 \in M_1$ and $s_{2_0} \stackrel{\mathcal{L}_{2_0}}{\longrightarrow} s_2 \in M_2$ and $\mathcal{L}_{1_0}, \mathcal{L}_{2_0}$ are *composable*.

$\overline{s_1} \sqsubset_{\mathrm{p}} s_2$, from induction hypothesis

From the definition of *path simulation*,

$\overline{s_1} \sqsubset_{\mathrm{p}} s_2 \Rightarrow \forall \overline{s_1} \stackrel{\overline{\mathcal{L}_1}}{\rightarrow} \overline{s_1'}, s_2 \stackrel{\mathcal{L}_2}{\rightarrow} s_2' \,.\, \overline{\mathcal{L}_1} \preceq \mathcal{L}_2 \Rightarrow \overline{s_1'} \sqsubset_{\mathrm{p}} s_2'$

Since, for every transition $\overline{s_1} \stackrel{\overline{\mathcal{L}_1}}{\rightarrow} \overline{s_1'} \in \overline{M_1}$, there is a transition $s_1 \stackrel{\mathcal{L}_1}{\rightarrow} s_1' \in M_1$,

$\forall s_1 \stackrel{\mathcal{L}_1}{\rightarrow} s_1', s_2 \stackrel{\mathcal{L}_2}{\rightarrow} s_2' \,.\, \overline{\mathcal{L}_1} \preceq \mathcal{L}_2 \Rightarrow \overline{s_1'} \sqsubset_{\mathrm{p}} s_2'$

Using lemma 5.5.1, $\overline{\mathcal{L}_1} \preceq \mathcal{L}_2 \Rightarrow \langle s_1, s_2 \rangle \rightarrow \langle s_1', s_2' \rangle \in T$

$\Leftarrow$:

To show the converse, $\langle s_1, s_2 \rangle \rightarrow \langle s_1', s_2' \rangle$ implies $\langle s_1, s_2 \rangle$ is reachable in M,

from the definition of *composition*. $\square$