FORMAL METHODS FOR EMBEDDED SYSTEM DESIGN

by

Harry Chia Chang Hsieh

Memorandum No. UCB/ERL M00/37

20 May 2000

FORMAL METHODS FOR EMBEDDED SYSTEM DESIGN

by

Harry Chia Chang Hsieh

Memorandum No. UCB/ERL M00/37

20 May 2000

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Formal Methods for Embedded System Design

by

Harry Chia Chang Hsieh

B. S. (University of Wisconsin at Madison) 1988
M. S. (Stanford University) 1991

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering-Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Alberto L Sangiovanni-Vincentelli, Chair
Professor Jan M. Rabaey
Professor Ilan Adler
Doctor Felice Balarin

Spring 2000

The dissertation of Harry Chia Chang Hsieh is approved:

Chair Date

Date

Date

Date

University of California at Berkeley

Spring 2000

Formal Methods for Embedded System Design

Harry Chia Chang Hsieh

Abstract

Formal Methods for Embedded System Design

by

Harry Chia Chang Hsieh

Doctor of Philosophy in Engineering-Electrical Engineering and Computer Sciences

University of California at Berkeley

Professor Alberto L Sangiovanni-Vincentelli, Chair

The goal of this dissertation is to investigate how formal methods can be applied to the domain of embedded system design. The emphasis is on the specification, representation, validation, and design exploration of such systems from a high level perspective. We start by reviewing the framework upon which the theories and experiments are based, and through which our formal methods are linked to synthesis and simulation. We also review in detail the formal model of computation used to represent embedded system specification and implementation: Network of Codesign Finite State Machines.

We formulate a formal verification methodology to verify general properties of CFSM networks and demonstrate that this methodology is efficient in dealing with the problem of complexity and effective in finding bugs. However, manual intervention is required. Manual intervention comes in the form of abstraction selection and separation of timing and functionality through constraints on the architectural mapping. We conjecture that for specific properties, efficient algorithms exist for completely automatic formal validation of systems.

The analysis of the application of formal verification techniques to CFSM networks led us to choosing a property that is basic, but embodies the principle of separation of timing and functionality: synchronous equivalence of two different implementations. This property is analogous to functional equivalence for sequential circuits. One powerful result of this equivalence criterion is the identification of a set of delay-insensitive scheduling policies. Once a delay-insensitive scheduling policy is chosen, any variation in delay does not affect the functional behavior. For efficient equivalence checking of implementations

using different delay-insensitive scheduling policies, we propose structural algorithms based on worst-case analysis of the communication among components. The events communicated between components are abstracted into a signature that is maximal in the sense that it represents all possible communication patterns of that system. By comparing the signatures of different delay-insensitive scheduling policies for a given system, we were able to determine equivalence conservatively.

Lastly, we relate communication analysis to exhaustive simulation through a series of refinement and pruning operations on the communication signatures. An algorithm can choose to work at any abstraction level trading off computational efficiency with the possibility of inconclusive result due to false negatives. We provide primitives to move between different abstraction levels that exist between communication analysis and exhaustive simulation. We demonstrate with real-life examples that synchronous equivalence opens design exploration avenues uncharted before.

Professor Alberto L Sangiovanni-Vincentelli
Dissertation Committee Chair

To my parents...

Contents

List of Figures

List of Tables

Acknowledgements

I am truly privileged to work alongside Prof. Alberto Sangiovanni-Vincentelli. His integrity, quality of work, and breadth of knowledge have never ceased to amaze me. In many ways he has inspired me, and will have a lasting influence on me. I am also grateful to Prof. Jan Rabaey, Prof. Ilan Adler, and Dr. Felice Balarin for providing useful comments for this dissertation. Dr. Balarin, more than anyone else, was there from the early stage of this research. He tirelessly fought in the trenches with me, resolving inconsistencies and removing misconceptions. For his guidance and constant encouragement, I am truly indebted.

Prof. Luciano Lavagno had given me invaluable help all through my stay here at Berkeley. It is not an exaggeration to say that without him, this dissertation would not have been possible. He helped to initiate many of the ideas and, being the principle architect of POLIS, provided me with a framework to experiment within. Berkeley CADgroup is a research environment most researchers can only dream about. I want to thank Prof. Robert Brayton, Prof. Tom Henzinger, Prof. Kurt Keutzer, Prof. Richard Newton, along with Prof. Sangiovanni-Vincentelli for their leadership and guidance that make CADgroup what it was, what it is, and what it will be.

I am blessed to be able to collaborate with many of the brightest minds in CAD community today. I want to give my sincere thanks to: Massimiliano Chiodo, Paolo Giusto, Attila Jurecska, Dr. Claudio Passerone, Roberto Passerone, Dr. Ellen Sentovich, Marco Sgroi, Kei Suzuki, and Dr. Bassam Tabbara. Many other CADgroup members, past and present, have helped me professionally or personally. I want to thank all of them. Some amongst them have to put up with me a little more than others: Luca Carloni, Wilsin Gosti, Ken Yamaguchi, Dr. Yuji Kukimoto, Dr. Rajeev Murgai, Dr. Rajeev Ranjan, Dr. Thomas Shiple, Suzan Szollar, Dr. Serdar Tasiran, Dr. Iasson Vassiliou, Dr. Tiziano Villa, Marlene Wang, and Dr. Yosinori Watanabe.

The staff support I received has been uniformly helpful and amiable. Thanks go to Kia Cooper, Ruth Gjerde, Brad Krebs, Elise Mills, Flora Oviedo, and Judd Reiffin. I would like to acknowledge the financial support of the Semiconductor Research Corporation. Most of this work is conducted under the auspices of SRC research funding (DC-324).

I want to thank the folks at University of California Ballroom Dancers and Dance Arts Studios. International Style Ballroom and Latin Dancing grew to become my second passion and primary social venue in the past few years. Many of the dancers have become

my close friends and confidants. They helped me to put aside all my troubles and worries, for as long as the music plays. I also want to thank Joanne Turnpenny and Jennifer Litty, who provided me with the greatest inspiration and much needed distraction.

My younger brother, Chia Hsing, has the distinction of being my only local family member for most of the past fifteen years. He helped me to reconcile much of my professional and social demands with my family obligations. I want to thank him both as a brother and as a friend. My older sister, Wan Ling, fueled much of my life-long passion for classical music. Finally, I owe everything to my father, Wen Ho Hsieh, and my mother, Chin Chi Wu Hsieh, who bore me, raised me, and showed me the true meaning of dedication, integrity, and compassion. All my past accomplishments serve witness to who they are, and all my future achievements will stand testimony to who they continue to be.

Chapter 1

Introduction

1.1 Emergence of Embedded Systems

Embedded systems are loosely defined as any system that utilizes electronics but is not perceived or used as a general purpose computer. Traditionally, one or more electronic circuits or microprocessors are literally embedded in the system either taking up roles that use to be performed by mechanical devices or providing functionality that is not otherwise possible. As prices of microelectronics and microprocessors continue to fall, it becomes increasingly attractive for systems to take on this electronic aspect and enjoy the advantage of higher performance, lower cost, additional features, flexibility to design changes, and faster time to market. Today, the low cost and high performance afforded by electronics spurs the creation of many products and systems that could not even be dreamed of just a few years ago. Some examples of embedded systems are automotive control systems, manufacturing systems, network switches, climate control systems, home appliances such as microwave ovens or refrigerators, cellular phones, Personal Digital Assistants, pagers, pacemakers, weapons, and toys.

This added electronic dimension allows the performance and cost of an embedded system to track the unbelievable improvement in the performance and cost of digital electronics. Traditional cellular phone transmission based on analog technology cannot compete in performance (clarity) with the cellular phone transmission based on digital technology and digital electronics. First generation microwave ovens often contained only a start/stop button and a timer. A consumer grade microwave oven today easily handles different settings for different food items, special functions such as reheating, defrosting, and browning,

and even contains sensors to sense the temperature of the food item for optimal result. Adding this electronic dimension to existing products also offers tremendous opportunities for improving the design. Electronic injection control in an automobile is an example where the added electronic dimension provides higher performance and lower cost. The production of traditional injectors needs to adhere to very stringent specifications to achieve high fuel efficiency and low emission. However, when powerful electronics are used to perform adaptive engine control, algorithms can be devised to control fuel injection taken into consideration also the condition of the injector. A cheaper manufacturing process can then be used to produce the injector and, at the same time, meet and exceed the requirements for fuel efficiency and exhaust emission.

The improvement in performance and cost can often be attributed to the use of electronic components that can be easily programmed or configured to perform various functions. All electronic components become more performing and cheaper as integrated circuit technology advances. Programmable and configurable components such as processors and gate arrays enjoy another dimension of economy of scale that is not available for components with very limited programmability. Since programmable and configurable components are able to perform a variety of functions, they can be applied to an ever-increasing variety of embedded applications. Successive generations of a product can also share the same components, or at least the same family of components, running different software or with a different configuration. As larger and larger portions of embedded functionality are performed on programmable components, the major cost factor of a design shifts from material and hardware design to the design of software programs. An effective embedded system design methodology must emphasize the design of software.

Another salient characteristic of embedded systems is their unconventional user and environment interfaces. Instead of keyboards and monitors, or even buttons and Light Emitting Diodes, the natural interfaces for embedded systems are more akin to sensors and actuators. Consider an engine control application, where the inputs to the system are shaft position and air and gas intakes, and outputs are control pulses for valves. Accelerometers are being inserted into pens today and embedded electronics are then used to gather authentication signatures. An effective embedded system methodology must take into consideration the design of the interfaces, and be very careful in defining the boundary of the electronic portion of the system.

The electronic aspect of embedded system can be implemented with many differ-

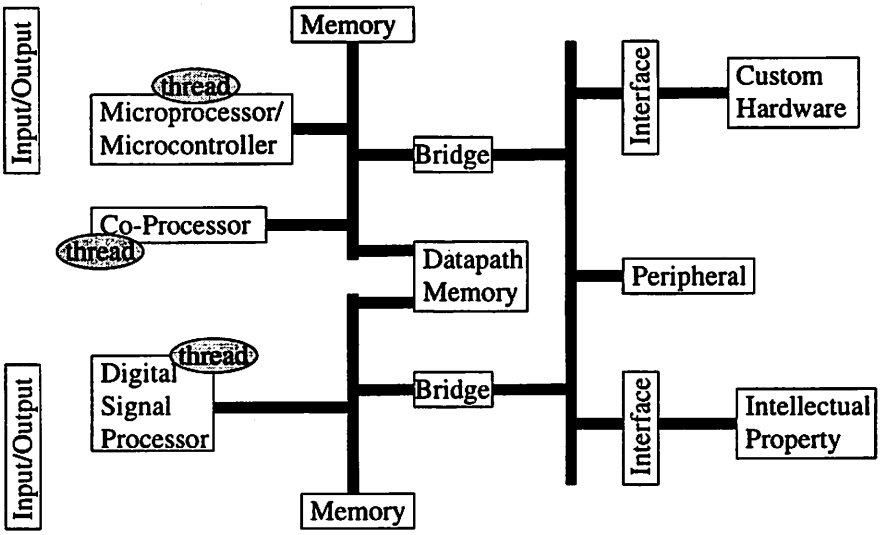

Figure 1.1: An Embedded System Architecture.

ent computational resources, ranging from off the shelf Small Scale Integration (SSI) and Medium Scale Integration (MSI) chips, to application specific devices like custom chips, semi-custom gate arrays, and field programmable gate arrays, to processors like Digital Signal Processors (DSPs), microprocessors, and microcontrollers. A possible hardware architecture for an embedded system is illustrated in Figure 1.1. The functionality of the system may be implemented on the microprocessor, the co-processor, and the DSP as threads. It may also be implemented by controlling the custom hardware and microcontroller peripherals via drivers. There may also be Intellectual Property (IP) blocks in the system such as MPEG decoders and ethernet interfaces. Each processor has its own program memory, data memory, and memory bus. The DSP additionally shares some datapath memory with the processors. Bridges to the custom hardware, peripherals, and IP help to interface these drastically different computational resources. The designer of embedded systems faces many significant challenges, including the obvious burden to possess expertise on software programming, hardware design, system design issues such as allocation of the functions to resources and scheduling of tasks, and interface design between different computational resources and to the rest of the system. Traditional simulation tools must be combined to simulate a system consisting of heterogeneous elements. Synthesis tools must synthesize interface and communication structures, and to some extent, software, if a high-level description language is used. There also must be a method to perform design

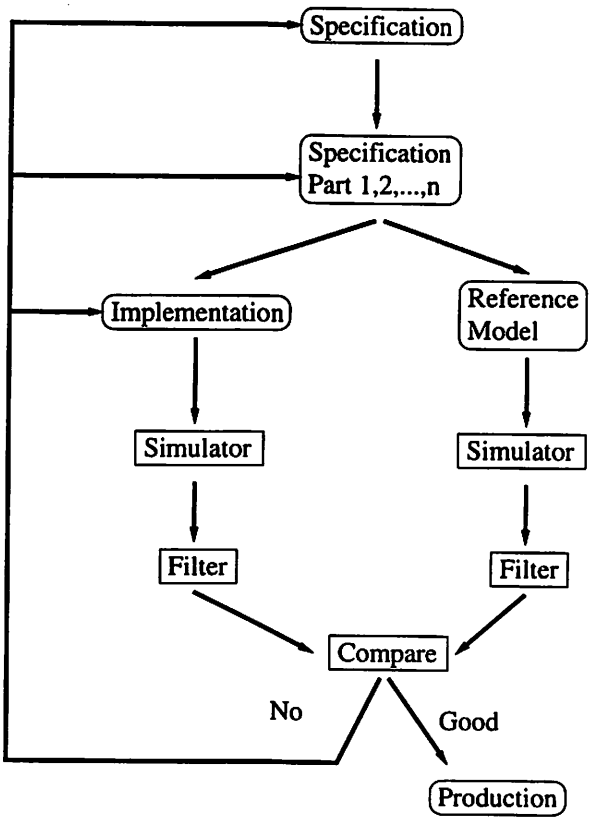

Figure 1.2: Current Embedded System Design Practice.

exploration, especially of the newly added dimension to the design freedom: a particular piece of computation may now be performed on any number of computational resources, with different performance characteristics.

1.2 Design of Embedded Systems

Current embedded system design practice, as illustrated in Figure 1.2, is quite informal and application specific. Designers often start with a requirement written in English, use "intuition" to pick a particular interpretation of this requirement and break the design into hierarchy so as to better manage the complexity of the design. Designers then take each part of this hierarchy and write a so-called reference (or golden) model in VHDL, Verilog, C, or any other language that has an execution semantics. The golden model is executed on a computer to investigate whether it satisfies a set of requirements, including

a match with the original informal specification. This is done on parts of the design and the result is extrapolated to the whole through simple logic deduction. A (candidate)[1] implementation is then generated through a combination of manual labor and tools that are often poorly connected. Different parts of the design may be manually mapped to different computational resources. Synthesis for different resources, as well as the interfaces between resources, usually utilizes tools that are based on completely different models of computation (e.g. finite state machines and data-flow networks), hence are very difficult to be connected in any way. The correctness and optimality of the (candidate) implementations are assessed with filtered simulation traces obtained from the reference model and from the candidate implementations. Simulation tools for different resources are also often disconnected and have very different models of computation. This contorted and highly informal design flow is obviously very error-prone. It does point out many fundamental difficulties in the design of embedded systems.

1.3 Requirements For An Effective Design Methodology

The design of embedded system is a highly complicated process. Due to this complexity in both size and characteristics of the problem, a push-button solution is highly unlikely for the near future. It is not reasonable to expect the embedded system designers to write down all the necessary detail of an embedded system specification, and only that necessary detail. It is also not reasonable to expect the tools to be able to optimize accordingly even if such a complex specification is written down. An effective methodology should instead provide designers with a seamless interactive flow from high level abstract specification all the way down to implementation. Central to this seamless flow is a formal model that can be used to represent both the abstract specification and the final implementation.

A representation is abstract when it is devoid of details. A mathematical equation is more abstract than a C program implementing the equation. The C program is more abstract than the object code compiled from the C program. In each case, the more refined representation contains more detail information than the more abstract representation. The C program contains algorithmic detail, which is absent in the equation. The object code contains memory requirements and delay characteristics, which are absent in the C program.

[1] An "implementation" generated through this process may only be considered a candidate because it may not be correct. It is not generated through formal refinement. Some *ad hoc* manual procedures are involved.

This absence of information has several important uses. First, it is easier for the designer to analyze a simple, abstract, representation. It is easier to read a C program to figure out what it is computing than to look at a piece of object code. Secondly, it is easier, computationally, for the tools to work on a more abstract representation. Reachability analysis of a set of equation implementing a sequential system is much more easily done on the equations (often in terms of Finite State Machines) than on the C program representing the sequential computation. The third important use of an abstract representation is that one can prove properties for the abstract representation that will hold also for the refined representation. Essentially, a refined representation is the abstract representation plus some detail. A property is proven for the abstract representation if it is proven regardless of what the resolution of that detail may be. It is therefore proven for every particular resolution, including the chosen refined representation. For example, if a set of iterative equations does not converge, then the C program should be non-terminating regardless of the algorithmic detail, and the object code will be non-terminating regardless of the delay characteristic and memory requirement.

Automatic tools are very important in supporting design decisions that may be done interactively. In an effective methodology, automatic tools should be used to provide the validation model for feedback to the interactive decision from the designer. Automatic synthesis tools are also used when the specification is refined enough, so that more manual intervention is not necessary. Only through formal specification, formal refinement, and automatic synthesis can one guarantee the correctness of a design.

Theoretically, it may be desirable to have a single "best" formal language for embedded system specification. Practically, this is very hard to achieve. Different types of function are often more efficiently and more naturally represented in different formal languages. In addition, how a specification is written often affects the output of an automatic synthesis tool. It is therefore more reasonable to have a set of formal languages for specification; all sharing a formal model which supports formal refinement. This will allow efficient synthesis because the designer can still write specifications in such a way that the given synthesis algorithm will produce the best result. Formal analysis is still possible because of the common formal semantics.

Design reuse, or Intellectual Property (IP) reuse, is another way to deal with the issue of complexity. Having a hierarchical specification written in multiple formal languages eases the process of IP integration. IP can be treated as just another hierarchy that has its

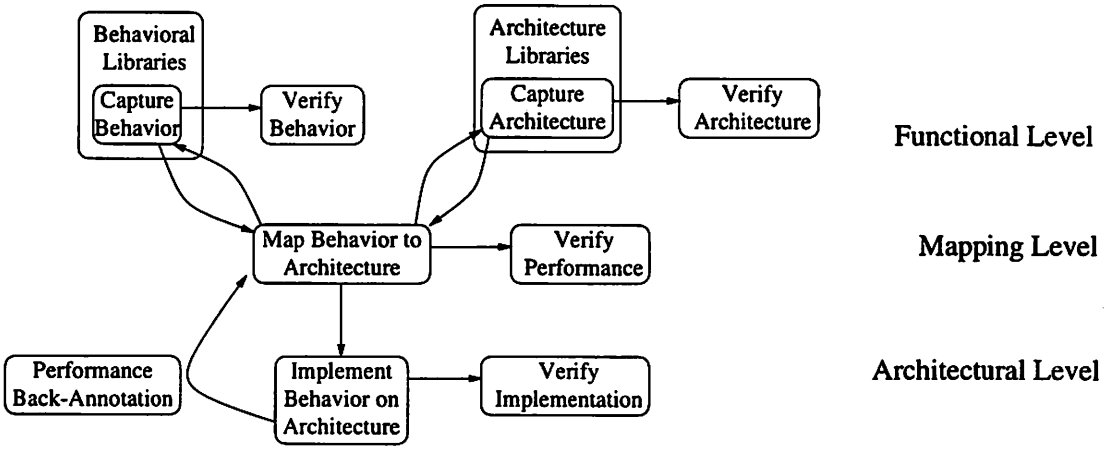

Figure 1.3: Proposed Design Approach.

own language and synthesis procedure and shares the same formal semantics. Characterization of the IP for analysis may be difficult, depending on the type of analysis required and the information available from the IP providers.

Another unique difficulty to the design of embedded system is in the choice of the target architecture. While conventional digital design problems usually focus on systems with a single, specific computational resource, embedded system design can often be realized using a variety of computational resources. To complicate the matter further, the "optimal" architecture for the application may consist of more than one type of computational resources. In fact, to satisfy the requirement and achieve the optimal design, the architecture needs to be designed, or "codesigned" alongside the algorithms. While it is not possible for any embedded system design framework to accommodate all possible architectures, it is important to accommodate a reasonable subset and make this restriction explicit so that the designer will not unknowingly over-constrain the design problem.

1.4 Proposed Design Approach

The overall design strategy that we envision is depicted in Figure 1.3. The design flow consists of three levels, from the conception of the design to its implementation. A single model of computation is used throughout, so the transition between different stages of the design is formal and precise.

At the functional level, behavior is described formally using a simple and concise

mathematical model. The goal is to impose as little implementation detail as is possible. A very large set of implementations may be the refinements of the specified behavior. In order to weed out some of these refinements which may have undesirable characteristics, a separate set of constraints may be needed to describe only those implementations that exhibit desirable characteristics, such as short timing delay and low power consumption. The relatively low complexity of the abstract behavior specification, however, makes it possible to perform abstract analysis on the behavior at this early stage of design process. Designers can perform property verification to see if the behavior exhibits certain critical properties and if it does not, modify the behavior or constrain the refinement process. The characteristic of a target architecture can be described by a set of succinct parameters. One such set may be described as a Motorola 68hc11 platform executing at 2MHz with an Introl compiler. While the most detailed information should be gathered and indeed may be indispensable for the final validation, abstract high level information, such as the "approximate" delay of a basic block, can be very useful for high level analysis given a particular abstract model of the behavior. Both the behavior and the architecture descriptions can be used for many different designs and is consistent with the philosophy of IP reuse.

The first few stages of the refinement from behavior specification are usually done manually and are collectively called the mapping level. The design space is simply too big at this level for any meaningful automation. Any refinement decision will eliminate a large set of implementations from consideration. It is more reasonable to allow the designers to use their intuition for these decisions. The primary functions of the tools will be to assist in making these decisions through analysis. Some examples of these mapping (also called architectural mapping) decisions include mapping behavior onto computational resources and choosing scheduling for shared resources. The types of interesting formal analysis at this level may include property checking for an abstract specification, or equivalence checking between two different implementations or between an implementation and a reference (golden) model.

After the architectural mapping stage, the synthesis tools have enough information to optimize the design all the way down to implementation. This includes technology-independent optimization, as well as actual instruction selection and register allocation for components mapped to programmable resources, and logic synthesis for components mapped to be implemented as circuits. To reduce the possibility of manual error, the synthesis procedure from this level down should be as automated as possible. The results of

synthesis are back annotated to the earlier part of the design process for possible iteration.

1.5 Motivation

We have developed a framework for the design of embedded system, POLIS, in which a single formal model of computation is used throughout the design process. Steps in the design process formally refine the abstract behavioral specification, synthesize the behavior down to implementation under some objectives or constraints, or transform the behavior to verification and simulation-oriented models. Due to the existence of this formal model, formal verification tools [BSVA+96, McM93] can be applied to verify properties at different levels of abstraction. The current generation of automatic formal verification tools suffers from the issue of complexity. Many large designs are too complex for automatic formal verification tools to analyze within reasonable time and memory resource. For the domain of control dominated embedded applications, we want to establish a verification methodology with systematic applications of abstractions and assumptions that will make it possible to verify large systems. We deal with this issue in Chapter 4. Unfortunately, even with the methodology in place, verification of arbitrary properties of arbitrary systems is still very tedious and involves many manual abstraction operations. We establish efficient, automatic, algorithms for formally analyzing specific, but important, properties of certain classes of embedded systems. The "property" that we choose in this work is the equivalence between two implementations of the same high level specification.

Current methods for specifying embedded system application are extremely *ad hoc*. The desired behavior is often loosely represented as some filtered simulation trace with special annotations, implicit assumptions, and so-called intuition. Therefore, a fundamental point of clarification to improve the design methodology is the formal definition of correctness. Because embedded system designs are inherently complex, the principle of "separation of concerns" in specification and verification is essential. Functional correctness and timing should be verified independently.

We thus define *synchronous equivalence*, a "functional" equivalence among a set of candidate implementations of embedded system specifications. An equivalence relation divides behaviors into equivalence classes. Equivalence analysis can answer the question whether two implementations are equivalent to each other or whether an implementation is equivalent to the "golden" model representing the specification. This can be done precisely

through state reachability analysis methods (e.g. model checking tools), or conservatively (but more efficiently) through structural methods. We derive efficient structural algorithms for synchronous equivalence analysis that can be used at the mapping level. These algorithms can be used to explore the design space efficiently. Since the algorithms are conservative, they could sometimes lead to design choices that are sub-optimal. Trade offs between the efficiency of the algorithms and the quality of the solution will need to be explored. We deal with the definition and properties of synchronous equivalence in Chapter 5. Analysis algorithms are presented in Chapters 6, 7, and 8, with different emphasis on the efficiency or the quality of the solution.

Once a behavioral specification is architecturally mapped according to some criteria, automatic synthesis can be done using any set of existing tools. SIS [SSL+92] or Design Compiler from *Synopsis* can be used for synthesis of components mapped to FPGAs. Esterel [BCG91] or Statechart [HLN+90] can be used for synthesis of components mapped to microprocessors. The POLIS codesign framework, however, provides a complete synthesis solution from behavioral specification all the way down to implementation with various computational resources, all consistent with a formal model of computation.

1.6 Thesis Overview

In the next chapter we review the specification, mapping, simulation, and synthesis aspects of the POLIS codesign framework [BCG+97]. These capabilities of POLIS complement nicely the formal methods that we develop in the later chapters. In addition, the formal model of computation used in POLIS, the network of Codesign Finite State Machines (CFSMs), will also be the formal model used to demonstrate the properties of the algorithms. CFSMs are discussed in detail in Chapter 3. After discussing the framework within which our formal methods will fit and the formal model we will use, in Chapter 4 we present an abstraction/refinement methodology for the formal verification of embedded systems. Extensive abstraction is used to handle the complexity of the design, and user-directed refinement is used as necessary to verify properties of the design. While our formal verification methodology is effective in verifying properties of a design, it requires extensive manual intervention. We develop automatic analysis algorithms for a specific, and extremely important, analysis: synchronous equivalence between two implementations of the same behavioral specification, or between an implementation and a reference model.

We define synchronous equivalence and the synchronous assumption in Chapter 5. We discuss the necessity of such an assumption and justify the resulting restriction on the design space. We also discuss the consequences of the definition of equivalence relation and the useful properties that come along with it. In Chapter 6, we present simple static algorithms that can be used to prove synchronous equivalence with low computational complexity, often with no more than simple checking of implementation attributes or simple traversal of system graphs. Unfortunately, an analysis of this sort is plagued by false negatives (pairs of implementations that are declared not equivalent while actually they). In Chapter 7, we propose efficient analysis algorithms for synchronous equivalence based on the observation that if the "communication" among components is the same for two implementations, then they must be synchronously equivalent. Communication analysis removes many false negative results from static algorithms, at a cost of higher complexity of the analysis. This effectively introduces a partial order verification strategy for CFSMs. In Chapter 8 we make our analysis method more precise by removing more and more of the false negative results. We show that communication analysis can be transformed into exhaustive simulation through the refining of "containers" and the pruning of unobservable events. We develop a whole range of algorithms that can work at different points on the abstraction/refinement scale, trading off the quality of solution with the complexity of the analysis. Finally, we conclude with some summary remarks and future directions in Chapter 9.

Chapter 2

The POLIS Codesign Framework

The POLIS codesign framework is a collection of related tools that help the designer to take behavioral specifications all the way down to actual implementations. The target applications of POLIS are control dominated embedded systems such as automotive control systems and ATM switches. There is no fixed target architecture *per se*. Designers are free to utilize architectures consisting of any number of computational resources connecting through simple interface generated automatically by POLIS. The designs in POLIS are implemented as Globally Asynchronous Locally Synchronous (GALS) systems. The designer breaks his design first into a set of hierarchical interacting networks. The lowest (leaf) elements in the hierarchy are extended finite state machines called Codesign Finite State Machines (CFSMs). These CFSMs also correspond to the smallest units that can be assigned to computational resources. Each CFSM is locally synchronous, while the interaction between CFSMs is asynchronous.

It is designer's responsibility to decide which CFSM should be mapped to which computational resource. If more than one CFSM is assigned to a computational resource, scheduling policy has to be specified to coordinate their execution. The process of assigning CFSM to resources and deciding scheduling policy for shared resources is called *Architectural Mapping*. A CFSM network, as initially specified by the designer, is abstract. Many possible behaviors, including different interleaving of executions of the CFSMs, are all consistent with the specification. An architecturally mapped CFSM network may have deterministic behavior (hence it is simulatable) if low-level synthesis directives are set to some deterministic (default) values. We believe that at this point in time, automation of the architectural mapping selection process is very difficult, and in fact, probably not desirable.

It is better to leave these aspects of design freedom to the designers' "intuition" and allow the designer to choose his own architectural mapping. Oftentimes, designers will even write their specification with a particular architectural mapping in mind. What POLIS provides is, once a particular architectural mapping is chosen, a quick path to prototype and simulation so the designers can test out their architectural mapping to see whether it actually matches the requirement. This is enabled by a procedure for system level co-simulation of heterogeneous computational resources.

Once a CFSM network has been specified and an architectural mapping has been chosen, POLIS can take over and automatically synthesize hardware, software, as well as any necessary interface. Components mapped to hardware are synthesized as sequential circuits that may be implemented on some semi-custom ASICs or FPGAs. Components mapped to software are first translated into reduced Control Data Flow Graphs called Software graphs, or S-graphs. Technology independent optimization is performed on S-Graph. Either object code or a subset-C code is generated. Subsequent technology dependent optimization is carried out by commercial compilers to obtain the final assembly codes for the processors. The event base interfaces between hardware and software computational resources are inserted automatically. Scheduling code is also automatically synthesized to realize the high level directives from the designer at the architectural mapping stage.

The underlying mathematical model for POLIS is a network of Codesign Finite State Machines (CFSMs). Any language with FSM semantics, such as Esterel [Ber96], State Charts [DH89], or the synthesizable subset of Verilog [BY93] and VHDL [Eng94], can be compiled into a locally synchronous CFSM. The global connectivity is specified either graphically in Ptolemy [BHLM90], or textually in a generic netlist. Since the communication between CFSMs is asynchronous with finite buffers, there is no guarantee of delivery of information. To ensure lossless communication, handshaking can be specified into the design itself. Alternatively and more efficiently, a scheduler can be used to prevent the loss of critical information. In Chapter 3, we will treat the subject of CFSM in full detail. Before that, we describe some relevant aspects of the POLIS codesign methodology.

2.1 The POLIS Codesign Methodology

One possible embedded system design flow is presented in Figure 2.1. The conception of the design, in the form of existing Intellectual Properties (IPs) and ideas, is written

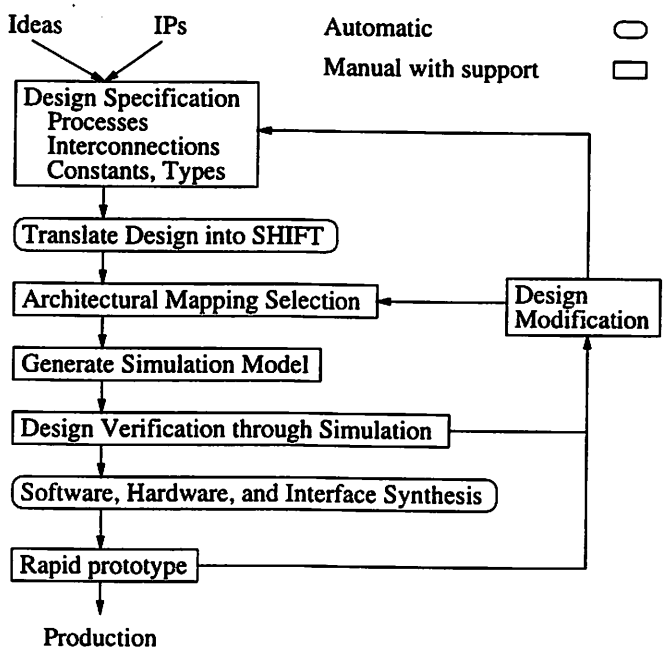

Figure 2.1: The Designer's Flow Chart.

down as a set of processes, interconnection and other design information. POLIS translates the processes into CFSMs, using the underlying FSM model of the specification languages. Along with connectivity and other information, the design is written into Software-Hardware Intermediate FormaT (SHIFT). After the designer selects an architectural mapping, POLIS generates quick or accurate simulation models so the designer can verify, through simulation, whether the design at this stage satisfies the functional and performance requirements. If it does not, the designer may want to "try out" another architectural mapping (and possibly different low-level synthesis directives). Once the combination of CFSM network, the architectural mapping, and the synthesis directives is deemed acceptable, the final implementation is automatically synthesized into actual codes for the target processors and the configuration files for gate arrays.

The complete POLIS codesign framework is represented in Figure 2.2 where ovals correspond roughly to tools and rectangles correspond roughly to models or files. Before we go through some important aspects of the framework, we present a very simple example that will serve to illustrate concepts throughout the rest of this dissertation.

Suppose that we want to specify a simple safety function of an automobile: a seat

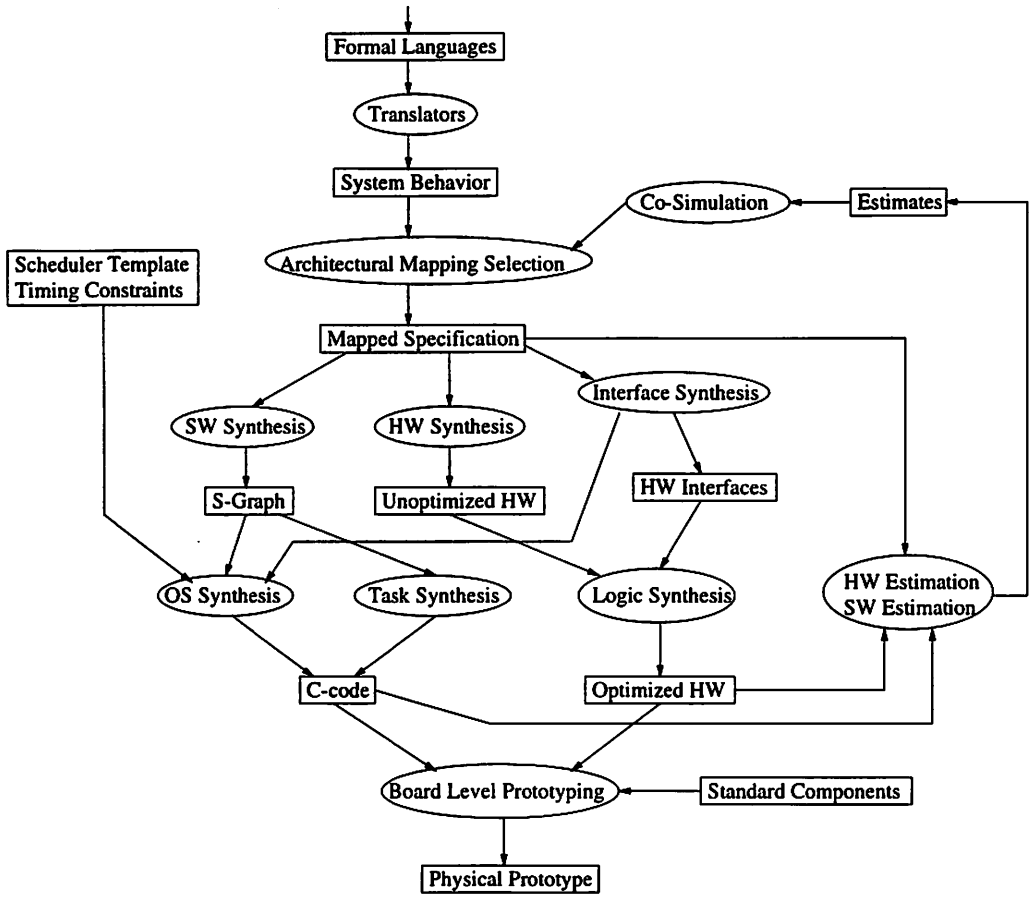

Figure 2.2: The POLIS System.

belt alarm control system. A natural language specification written by a designer could be: "Five seconds after the key is turn on, if the belt has not been fasten, an alarm will beep for five seconds, or until the key is turn off." The specification can be represented by two reactive components as shown in Figure 2.3, consisting of two CFSMs: A controller and a timer. CFSMs representing the controller and the timer are shown in Figure 2.4 and Figure 2.5, respectively. Input and output for a state transition are separated by "/". Conjunction is represented by "*", disjunction by "+", and negation by "!".

2.1.1 High Level Language Translation

In POLIS, designers write their specification in a high level language. Any high level language with precise semantics and underlying FSM model can be accommodated.

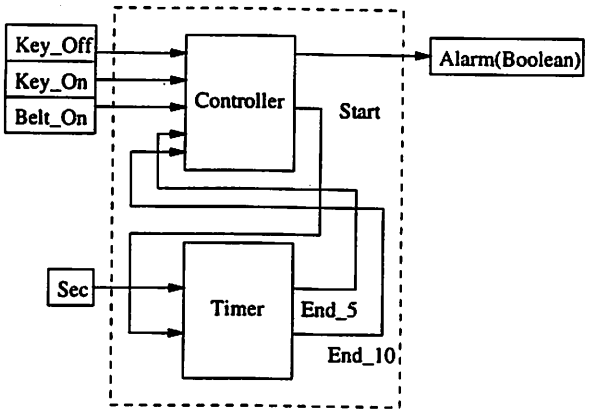

Figure 2.3: Seat Belt Alarm Controller Example.

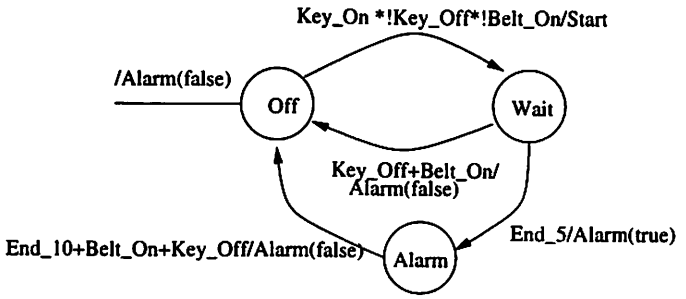

Figure 2.4: Controller FSM in Seat Belt Example.

Examples of the accepted languages include synthesizable subset of Verilog [BY93] or VHDL [Eng94], StateChart [DH89] or any other graphical FSM formalism, or Synchronous Languages such as Esterel [Ber96] and Lustre [HCRP91]. Another "netlisting" language is needed to describe the interconnection between these locally synchronous components. It can be in the form of a simple text file or compiled from the graphically interface provided by Ptolemy [BHLM90].

The seat belt controller can be specified as an Esterel text file as follows (the transition triggered by Reset did not appear in the graphical CFSM for the sake of simplicity):

```
module controller:
input Reset, Key_On, Key_Off, Belt_On, End_5, End_10;
output Alarm(boolean), Start;
loop
  do
    emit Alarm(false);
```

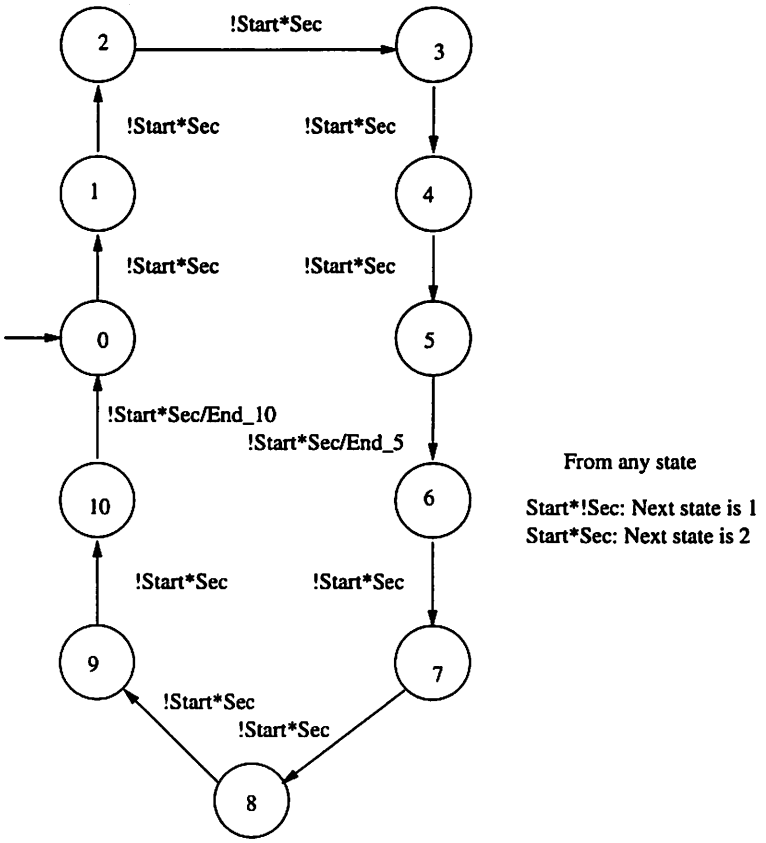

Figure 2.5: Timer FSM in Seat Belt Example.

```
    every Key_On do
      do
        emit Start;
        await End_5;
        emit Alarm(true);
        await End_10;
      watching [Key_Off or Belt_On];
      emit Alarm(false);
    end
  watching Reset
end.
```

The first two lines declare the input/output of the locally synchronous component. All the input signals are *control* signals. They are called *pure* signal in Esterel. There are no data portions associated with these signals. Output signal `Alarm` has a control portion, and a data portion that is associated with a Boolean value. The `loop` statement loops forever.

This is necessary to make the locally synchronous component responsive to the inputs at each "clock". Program within `Do ... watching Reset` will be executed till termination unless it is preempted by the occurrence of `Reset`. The `every Key_On` statement executes its body every time `Key_On` occurs. The `await End_5` statement stops the program and waits for the occurrence of `End_5` signal. It identifies a state of the system. In this case, state `Off` corresponds to `every Key_On` (that implicitly awaits `Key_On`), state `Wait` corresponds to `await End_5`, and state `Alarm` corresponds to `await End_10`. The `Emit` statement is used to specify that an output is present in a given transition. It can also be used to write the value associated with the signal. The Esterel file associated with the timer is as follows:

```
module timer:
constant Count_5:integer;
input Sec, Start;
output End_5, End_10;
every Start do
  await Count_5 Sec;
  emit End_5;
  await Count_5 Sec;
  emit End_10;
end.
```

The second line declares a symbolic constant whose actual value will be defined elsewhere. The `await Count_5 Sec` statement counts `Count_5` transitions in which signal `Sec` has an event.

A CFSM component is locally synchronous. This means that each component takes in a set of inputs, performs a computation and emits outputs. Upon the completion of the computation, it takes in another set of inputs. Since the communication among CFSMs are asynchronous, other CFSM and the environment may send information to a CFSM which may still be executing. This information will be kept in a one-place buffer. It is through this buffer that CFSM network decouples the locally synchronous computation from the globally asynchronous communication. It also facilitates using synchronous language, with its synchronous hypothesis, at the local level. There are no blocking writes, so the sending CFSM simply emits its outputs and continues execution. The buffer may overflow if the sending CFSM or the environment is too fast for the receiving CFSM. Requiring blocking write for all communication will make the implementation too costly. To prevent loss of information, synchronization can always be written into the design (e.g. hand-shaking), or set as a requirement on the implementation (e.g. scheduling policy).

CFSM networks will be described in more detail in Chapter 3.

2.1.2 Architectural Mapping

Architectural Mapping, to be defined formally in Chapter 6, is the process of selecting a particular target architecture, assigning CFSM components into computational resources in the architecture, and selecting schedulers if there are shared resources in the architecture. This decision is based heavily on design experience and is very difficult to automate. POLIS provides the designer an environment to quickly make any such decisions through various design hints from system co-simulation, and to quickly evaluate the resulting implementations.

The behavior of a specification is determinized in several stages. At the specification level, the CFSM network specifies the structure of the design (i.e. I/O connectivity of the components) and the functions of the components (i.e. transition and output functions of individual CFSMs). A CFSM network specification by itself is non-deterministic. More than one behavior is possible and many implementations can be consistent with a CFSM network specification. The designer implements the CFSM network by selecting the architectural mapping of the CFSM network. An architectural mapping makes the CFSM network deterministic by additionally specifying:

- (Possible) Delays of component executions.
- Scheduling policy, i.e. the conditions for a component to be enabled for execution, as well as the conditions for an enabled component to actually be executed.

Assigning CFSMs to particular computational resources limits the possible delays and scheduling policies that a CFSM network may have. They can be completely determinized when both the low-level synthesis directives (e.g. choice of compilers and compiler options) and schedulers for shared resources are specified. Some typical real-time scheduler for processors includes Cyclic Executive Serial (CES) scheduler, where tasks are executed in a fixed cyclic order that may contain repetitions, and Static Priority Serial (SPS) scheduler [LL73, ABD+95], where order of execution is based on a priority in that the task to execute is dynamically chosen according to a statically determined priority order. Though not often considered a scheduler for processors since it requires parallel resources, the Unit Delay Parallel (UDP) scheduler executes all tasks in parallel with the same delay. Com-

ponents mapped to hardware circuit resources such as gate arrays can be implemented as having an UDP scheduler.

A CFSM network can be simulated only if it is deterministic (i.e. architecturally mapped and synthesis directives set). We refer to a specification with an architectural mapping and a set of synthesis directives as an implementation of that CFSM network specification.

2.1.3 System Co-Simulation

Simulation of a system with heterogeneous computational resources is generally performed with separate simulation models. This makes trade-off evaluation difficult because the models must be re-compiled whenever a change in the mapping is made. System co-simulation is a way to give designers early feedback on their mapping choices. Early co-simulation of the entire system is possible in POLIS because of its simplifying delay assumption, automatic software synthesis, and performance estimation based on characterization of possible target architectures. This technique is, in practice, almost cycle-accurate, and uses the same model for all types of components. Only the timing information needs to be changed for any change in mapping.

In system co-simulation, synthesized C code is used to model all the components of a system, regardless of their future implementation. It is synthesized from an initial specification written in a formal language that can be mapped to different computational resources. Different implementation choices are reflected in different simulated times required to execute each task and in different execution constraints (e.g. mutual exclusion for embedded software).

Each event occurring in the system is tagged with a time of occurrence. The time of internal events, emitted by CFSMs in response to external or other internal events, is determined by using an estimate of the timing characteristics of the CFSM executions. This estimate is obtained by using a timing estimator to analyze the CFSM for speed characteristics. The algorithm uses a formula, with parameters obtained from running benchmark programs on the particular architecture, to compute the delay of each execution. If more precise timing information is needed, an instruction set simulator can be used [LLSV99].

Figure 2.6 contains a fragment of the synthesized C file for co-simulation of the seat belt controller. Figure 2.7 describes the complete code structure. One can see, for example,

the check for the presence of the Reset signal, near the top, and the communication of the Alarm signal near the bottom of Figure 2.6. Variable v_alarm carries the value of the Alarm event, which is emitted by emit_e_alarm (emit alarm in Figure 2.7). The state encoding is 0 for the initial (transient) state, entered when the CFSM starts its operation, 1 for ALARM, 2 for OFF and 3 for WAIT. The DELAY macro accumulates the timing information for each basic block. At simulation time, the executed code accumulates timing information for the chosen processor. The auxiliary code that is loaded as part of the Ptolemy simulation environment uses that timing information, together with the user-selected architectural mapping and synthesis directives in order to assign an appropriate time stamp to every simulation event. For hardware CFSMs, the delay for every transition is just one time unit (clock cycle).

2.1.4 Synthesis

A CFSM sub-network chosen to be implemented as a sequential circuit is directly mapped into an abstract hardware description format, BLIF [SSL+92]. POLIS implements each transition function as a combinational circuit, optimizes it using logic synthesis, and latches some circuit outputs to implement state variables. A CFSM sub-network chosen to be implemented on a processor is mapped into a software structure that includes a procedure for each CFSM, together with a simple Real-Time Operating System (RTOS). The CFSMs are synthesized through a two step process: A technology independent phase where optimization is done on a restricted control data flow graph called S-Graph [CGH+99], and a technology dependent phase where compiler are used to optimized for a particular processor configuration and instruction set. The RTOS is generated according to the architectural mapping chosen.

Interfaces between different computational resources are automatically synthesized within POLIS. These interfaces are in the form of cooperating circuits and software procedures (I/O drivers) embedded in the synthesized implementation. Together they implement the chosen architectural mapping as well as event communication and one-place buffer.

```
/* defines event emission and detection macros */
#include "os.c"
...
extern unsigned_int8bit v_alarm;
...
void _t_z_belt_control_0(int proc, int inst)
{
    static unsigned_int8bit v_state_tmp;    /* buffer of v_state */

    v_state_tmp = v_state;   /* update the state variable buffer */
    startup(proc);           /* update the input event buffers */
    DELAY(69);
L1: if (detect_e_reset_to_z_belt_control_0) { /* did reset occur ? */
        DELAY(40); goto L4;
    }
    else {
        DELAY(26); goto L2;
    }
...
L3: switch (v_state_tmp) {        /* branch based on current state */
    case 1: DELAY(41); goto L6;
    case 0: DELAY(63); goto L4;
    case 3: DELAY(85); goto L14;
    default: DELAY(107); goto L7;
    }
...
L10: DELAY(12); v_state = 2; goto L0;       /* go to next state */
L11: DELAY(10); v_alarm = 1; goto L12;      /* alarm is true */
L12: DELAY(14); emit_e_alarm(); goto L13;   /* emit event */
...
end:
    always_cleanup(proc);        /* clear input event buffers */
    return;                      /* return to RTOS/simulator */
}
```

Figure 2.6: C Code Synthesized for the Seat Belt Alarm Controller.

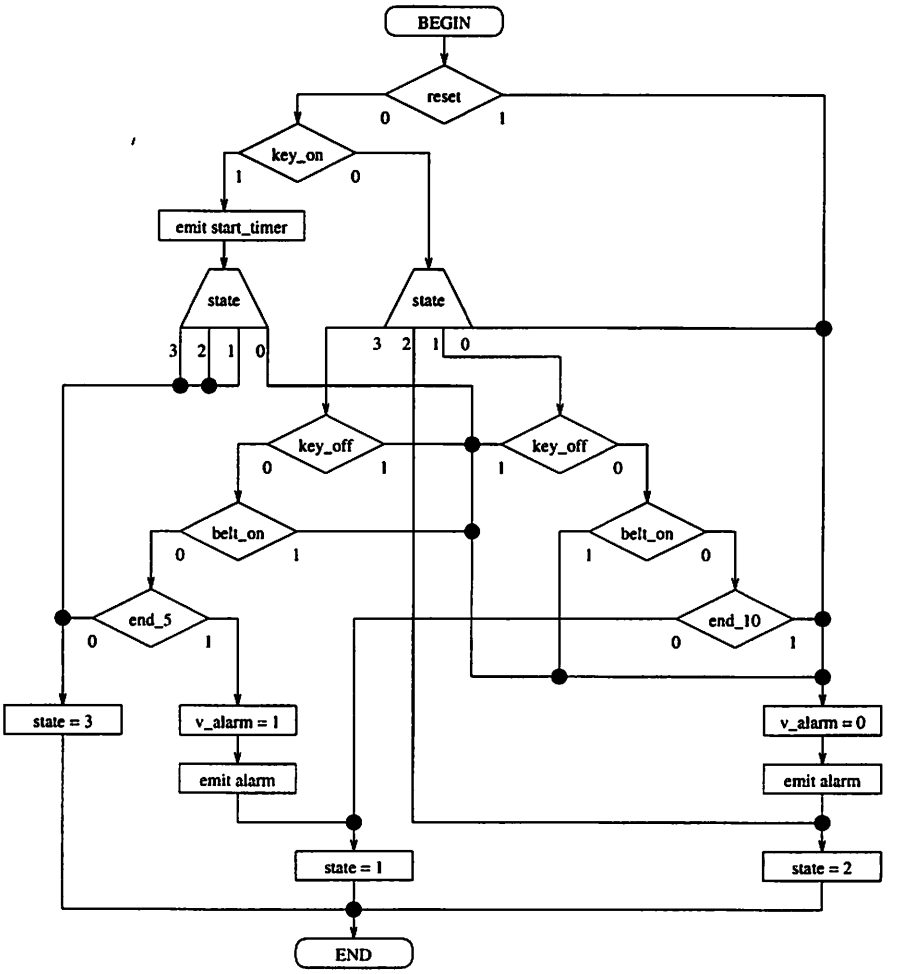

Figure 2.7: The Control and Data Flow Graph of the Seat Belt Controller.

Chapter 3

Codesign Finite State Machines

This chapter describes in detail Codesign Finite State Machines (CFSMs), the formal model used in POLIS for specification, simulation, analysis, synthesis, and optimization. It is also the model of computation that we used to demonstrate our formal verification methodology and equivalence analysis algorithms.

In Section 3.1, we first provide background information on hierarchical process network, finite state machines, and extended finite state machines. We present an intuitive view of the operation of the CFSM and CFSM network in Section 3.2. Through such intuitions, we justify some of the choices in designing the model of computation. In Section 3.3 we present a rigorous formal semantics of CFSMs and CFSM networks in terms of finite automata.

3.1 Background

A network of CFSMs can be seen as a hierarchical process network composed of extended finite state machines.

3.1.1 Hierarchical Process Network

A hierarchical process network is a structural model that consists of a set of *nodes* representing *processes* and/or *sub-networks*, and a set of edges representing *communication links*. Its semantics is determined by a node model and a communication model.

Each node of a network can be:

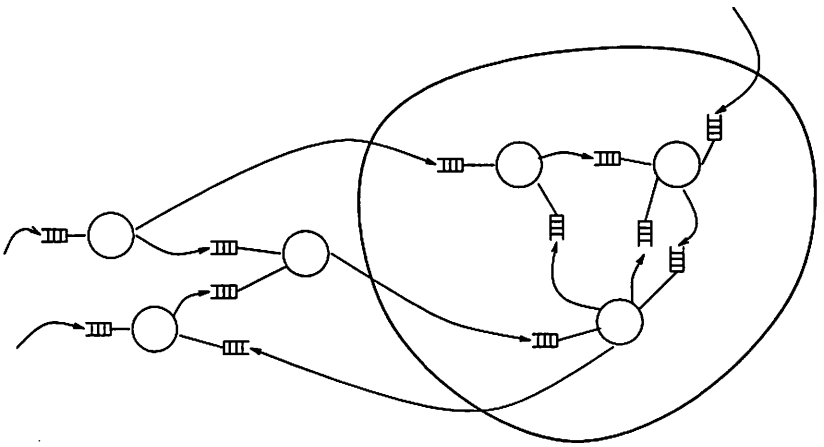

Figure 3.1: Hierarchical Process Network.

1. A process ("leaf" node), which reads data from incoming edges, computes a function based on those data and writes information along outgoing edges. This process may be represented by a Finite State Machine, a Control/Data Flow Graph, or some code in an imperative language (e.g. C, JAVA). In the case of network of CFSMs, the process is represented by an extended finite state machine.

2. An instance of a network. The instantiation mechanism can be *static* if the total number of nodes in a given hierarchy is fixed, or *dynamic* if it is variable. Dynamic instantiation may increase the expressive power of the model, while static instantiation is only a convenient mechanism to reduce the size of the representation. An appropriate mechanism to dynamically update the communication structure must also be provided for dynamic instantiation. In the case of network of CFSMs, only static instantiation is allowed.

The communication can be generically represented by a uni-directional queue which sits at the input of the receiver[1]. Communication among processes involves *writing* to and *reading* from queues. Sometimes just the act of reading or writing is used for synchronization purposes, and the transmitted information is ignored. We can characterize communication according to the following two aspects:

- *size of the queue*: fixed, bounded, or unbounded.

[1]In this discussion, we can ignore the size of the data stored in each queue, and assume them to be equal for the sake of simplicity.

- *synchronized/unsynchronized read/write*: a communication link has the synchronized read characteristic if the number of read operations at any point in time must be less than or equal to the number of write operations up to that point (i.e., each datum is read at most once). A communication link has the synchronized write characteristic if the number of write operations minus the number of read operations is less than or equal to the queue size (i.e., no data is lost by an overflowing queue). The synchronization characteristics are usually imposed by an OS, a scheduler, or a controller that is part of the definition of the formalism itself.

Synchronized and unsynchronized writes are equivalent in the case of unbounded queues.

A shared variable is an example of a communication method with unsynchronized read, unsynchronized write, and queue of fixed size 1. Reactive systems where tokens are consumed have synchronized read, but may or may not have synchronized write, depending on whether or not overwrite is allowed to happen. Some flavors of data-flow graphs have synchronized read, unsynchronized write over unbounded queues as a specification model ([Kah74, Den75]). Execution of a data-flow network in an embedded system generally assumes (tightly) bounded resources. A central problem for the implementation of a given data-flow network is the definition of a schedule (order of execution for its actors) that guarantees bounded queue size ([BHLM90, Buc93]). In the case of a CFSM, the communication method chosen is synchronized read, unsynchronized write over the queue of fixed size 1.

3.1.2 Finite State Machines

Finite State Machine (FSM) is a popular model for embedded system specification, due to the wealth of analysis and synthesis techniques that are available. In their most basic form they are suited mostly for simple protocols, and need an extension (described below) with integer variables and operations to efficiently represent real-life designs.

Definition 3.1 (Finite State Machine) *A* Finite State Machine *(*FSM*) is a quintuple* $\mathcal{F} = (I, O, X, R, F)$*:*

- I *is a finite set of input symbols.*
- O *is a finite set of output symbols.*
- X *is a finite set of states.*

- $R \subseteq X$ *is the set of initial states.*
- $F \subseteq I \times X \times X \times O$ *is the transition relation.*

$\mathcal{F}$ is *completely specified* if for all $i \in I$, $x \in X$ there exists at least one $x' \in X$, $o \in O$ such that $(i, x, x', o) \in F$, i.e. if the FSM has *at least* one choice of next state/output for each input/present state combination.

$\mathcal{F}$ is *deterministic* if R is a singleton and F is a *function* $F : (I \times X) \mapsto (X \times O)$, i.e., if the FSM has *at most* one choice of next state/output for each input/present state combination.

In general, a FSM that is implemented is forcibly *deterministic* and *completely specified*. An implementation usually has predictable behavior and has some (possibly empty) output.

3.1.3 Extended Finite State Machines

The *Extended Finite State Machine* (EFSM), or *Finite State Machine with Data path* (FSMD) [GR94] model is similar to the FSM model, but the transition relation may also depend on a set of internal variables. For notational convenience, the transition relation of an EFSM is usually described using relational, arithmetic, and Boolean operators. If the ranges of these variables are bounded, the set of possible states of the EFSM is still finite and its power is still within regular languages. If the ranges of these variables are unbounded integer or real, then the descriptive power is equivalent to that of Turing machines [HU79] and the state space is no longer finite. The EFSM model is traditionally used in hardware design, via the so-called "synthesizable subsets" of HDLs such as VHDL [Eng94] or Verilog [BY93]. The communication between EFSMs is usually through handshaking, which is synchronized read, synchronized write on a queue of fixed size 0.

EFSMs suffer from the so-called *state explosion* problem, because their composition is described synchronously (queue size 0). This implies instantaneous communication, which is difficult to implement efficiently in a distributed environment. For example, if one EFSM is implemented in hardware and the other in software, the EFSM composition mechanism requires that every transition be taken simultaneously by both. This is clearly inefficient, because hardware is now required to run at the same speed as the software. Moreover, EFSM composition is

- either non-responsive, if the EFSMs are Moore machines, in which outputs are delayed by one cycle with respect to inputs. Moore EFSMs are more difficult to use than Mealy machines for reactive design.

- or *possibly* non-causal, if the EFSMs are Mealy machines, in which outputs can appear in the same cycle as the inputs that caused them. This means, intuitively, that combinational feedback loops *can* happen when composing Mealy EFSMs, and hence the correctness of the composition must be checked on a case-by-case basis. Checking this correctness is a central problem also for compilation of synchronous languages, whose semantics is defined by composition of Mealy EFSMs ([BCG91, SBT96]).

Network of CFSMs utilize EFSM only at the leaf node, and do not necessarily implement synchronous communication like EFSM networks generally do.

3.2 CFSMs: Semantics

Implementations of the same piece of computation on different computational resources should result in the same "functional" behavior, though they may be drastically different in execution and communication time, as well as other non-functional characteristics such as power and cost. We extend the FSM model to include a communication mechanism that captures the essence of the timing behaviors of different implementation styles. This communication mechanism is based on asynchrony. The CFSM network model consists of extended FSMs that communicate among themselves asynchronously, which allows the expression of the essential notion that communication and computation take time.

A system is described as a network of CFSMs. Each CFSM is an extended FSM, where the extensions add support for data handling and asynchronous communication. In particular, a CFSM network has

- a *finite state machine part* that contains a set of inputs, outputs, and states, a transition relation, and an output relation.

- a *data computation part* in the form of references in the transition relation to datapath computation.

- a *locally synchronous behavior*: each CFSM executes a transition by producing a single output reaction based on a single, snapshot input assignment in zero time.

Once a reaction starts, it must continue to the end before the next reaction to some other input assignment can start. This is synchronous from *its own perspective.*

- a *globally asynchronous behavior*: each CFSM reads inputs, executes a transition, and produces outputs in an unbounded but finite amount of time as seen by the rest of the system. This is asynchronous communication from the *system perspective.*

This semantics, along with a scheduling mechanism to coordinate the CFSMs, provides a GALS communication model: Globally (at the system level) Asynchronous and Locally (at the CFSM level) Synchronous.

A CFSM network is a hierarchical process network with static instantiation. A process in a CFSM network (i.e. a single CFSM) is a flavor of reactive Finite State Machine extended with data path computation. There is a large body of knowledge about FSM models both in hardware and in software design. Having an FSM-based model gives easy access to existing synthesis and validation algorithms. In addition, the POLIS design methodology targets control-dominated applications, which are well suited for specification with FSMs. The CFSM network communication is characterized by synchronized read (for control signals), unsynchronized write over a queue of fixed size 1. Synchronized read is essential in implementing the "reactiveness" of the processes. In reactive systems, processes only execute when there is something on their input ports. Unsynchronized write is necessary with a bounded queue so that the sender can continue execution without waiting for the receiver. This leads to a more efficient implementation than the full handshake required by a synchronized write over a bounded queue. If communication over some specific link cannot be lost due to overwriting, this synchronized write requirement can still be specified either as a requirement on the scheduler, or by explicitly modeling a longer queue or a handshake as part of the design.

3.2.1 Signals

CFSMs communicate through *signals.* A signal is an associated pair: an *event* that is Boolean, and *data* from an integer subrange. A signal is communicated between two CFSMs via a *connection* that has an associated *1-place input buffer.* The buffer contains one memory element for the event (event buffer) and one for the data (data buffer). A sender CFSM always writes the data first and then emits its event. This order ensures that the data is ready when the event is communicated.

CFSMs *initiate* communication through events. The input events of a CFSM determine when it may react. That is, the model forbids a CFSM to react unless it has at least one input event present (except for the initial reaction) Without this restriction, a global clock would be required to execute the CFSMs at regular intervals, and this clock would in fact be a triggering input for all CFSMs. This would imply a more costly implementation.

3.2.2 Process Behavior

The functional behavior of a CFSM at each execution is determined by the specified *transition relation* (TR). This relation is a set of tuples: ($F \subseteq I \times X \times X \times O$). (refer to Section 3.1.2). Each tuple of the TR represents a specified *transition* of the machine, and the set of tuples is the specified behavior of the machine.

At each execution, a CFSM

1. Reads an input assignment and consume the inputs (setting input event buffers to 0)
2. Looks for a transition such that the read inputs I and the present state of the CFSM matches the transition relation.
3. If **Transition** is found, it is executed by
   (a) making the state transition to next state.
   (b) writing the new output values and events.
4. If **Transition** is not found, the CFSM returns control to its scheduler.

Each state variable may have a designated set of initial values that are specified with the transition relation. A set of initial values, one for each state variable, is a *initial state.* The *initial transition(s)* is a special transition(s) where the present state part is equal to the reset state. Moreover, this transition is allowed to not have any input events present in the input assignment.

3.2.3 Network Behavior

A *net* is a set of connections on the same output signal, i.e., it is associated with a single sender and at least one receiver. There is an input buffer for each receiver on a

net, hence the communication mechanism is *multi-cast*: a sender communicates a signal to several receivers with a single emission, and each receiver has a private copy of the communicated signal. Each CFSM can thus independently detect/consume and read its inputs.

A *network* is a set of CFSMs and nets. The behavior of the network depends on both the individual behavior, and that of the global system. In the specification model, the system is composed of CFSMs and a scheduling mechanism coordinating them. The scheduling mechanism at the specification level is non-deterministic and allows all possible interleaving. The only restriction imposed is the locally synchronous behavior, i.e. once an execution of a CFSM begins, it must execute to completion before the next execution of the same CFSM can start. The scheduling mechanism in the specification model may take several forms in the implementation: a simple RTOS scheduler for software on a single processor and concurrent execution for hardware, or a set of RTOSs on a heterogeneous multi-processor for software and a set of scheduling FSMs for hardware.

Each CFSM execution can be associated with a single *transition point*, t_i. t_{i+1} denotes the next transition point of the same CFSM. The model dictates that it is at this point that the CFSM begins reacting: reading inputs, computing, changing state, and writing outputs. Since the reaction time is unbounded, one cannot say exactly at which time a particular input (event or data) is read, at which time that input had previously been written, or at which time a particular output is written. There are, however, some restrictions. For each execution, each input signal is read at most once, each input event is cleared at every execution, and there is a partial order on the reading and writing of signals. Since the data value of a signal (with an event and data part) only has meaning when that signal is present, the model dictates that the event is read before the data. Similarly for the outputs, the data is written before the event, so that it is valid at the time the event is cast. This means that for transition point t_i,

- an input may be read at any time between t_i and t_{i+1} (but not later, because that would correspond to transition point t_{i+1}),
- the event that is read may have occurred at any time between t_{i-1} and t_{i+1},
- the data that is read may have been written at any time between t_0 and t_{i+1}, and
- the outputs are written at some time between t_i and t_{i+1}.

After reading an input, its value may be changed by the sender before t_{i+1}, but the receiver reacts to the captured input and the new value is not read until the next reaction.

3.3 Mathematical Model

In this section we give a formal definition of CFSMs and their semantics in terms of finite automata. Finite automata have precise mathematical semantics and are therefore amendable to formal analysis of many kinds including formal verification.

3.3.1 Preliminaries

3.3.1.1 Finite Automata

Finite automata [HU79] are structures used to define a (possibly infinite) set of sequences over some alphabet Σ. Formally, an automaton is a triplet (S, I, T) where S is some finite set of *states*, $I \subseteq S$ is the set of *initial states*, and $T \subseteq S \times \Sigma \times S$ is the *transition relation.*

An infinite sequence $s_0, s_1, \ldots$ of states, where $s_i \in S$, is said to be a *run* of a sequence $\sigma_1, \sigma_2, \ldots$ of elements of Σ, if s_0 is an initial state, and for all $i \geq 1$ the triplet (s_{i-1}, σ_i, s_i) is in the transition relation. The *language* of an automaton is the set of all sequences of elements of Σ that have a run. Both the sequence and the set may be infinite. Automata are very powerful tools to describe infinite set of infinite sequences with concise descriptions.

A *composition* of two automata (S_1, I_1, T_1) and (S_2, I_2, T_2) over the same alphabet Σ is the automaton:

$$(S_1 \times S_2,\ I_1 \times I_2,\ \{((s_1, s_2), \sigma, (q_1, q_2)) \mid (s_1, \sigma, q_1) \in T_1 \wedge (s_2, \sigma, q_2) \in T_2\}) \ .$$

where $s_1, q_1 \in S_1$, and $s_2, q_2 \in S_2$. The language of the composition of two automata is the intersection of their languages.

With no loss of rigor, we can assume that the alphabet Σ is the set of assignments to *alphabet variables*, which can take on some values. This allows a clearer connection with Finite State Machines. Automata can be represented graphically as a *state transition graph* (STG), or in a textual format. The two representations (textual and STG) are equivalent, but STGs are more convenient for controller-like automata, and textual representations may be more compact for data-path-like automata.

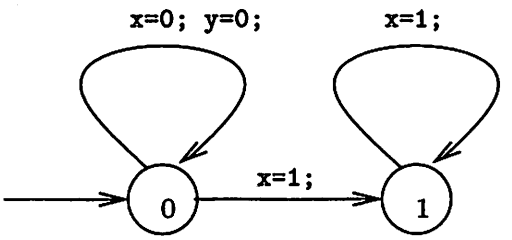

Figure 3.2: The STG of an Automaton.

For STG, the nodes correspond to states of the automaton, and initial states are marked by arrows. The edges are labeled with predicates over alphabet variables, such that σ satisfies a predicate p on the edge $s \rightarrow q$ if and only if $(s, p(\sigma), q)$ is in the transition relation.

Consider, for example the STG in Figure 3.2, and assume that the alphabet variables x and y are both binary valued. Note that if a predicate does not appear on the edge, it is always satisfied.

One can easily check that that the sequence:

x	0	0	1	1	1	1	...
y	0	0	1	0	1	0	...

is in the language of the automaton represented in Figure 3.2, because it has a run 0 0 0 1 1 1 1 ....

A sequence that does not have a run would be

x	0	0
y	0	1

The state sequence corresponds to this input sequence begins with state 0 (initial state), transitions to state 0 on $x, y = 0, 0$, and then is blocked. In state 0 with input $x, y = 0, 1$ there is no specified transition. Hence, the input sequence does not have a run.

Alternatively, we can use a textual form to represent a transition relation. We view states as possible assignments to *present state* and to *next state* variables. We assign an arbitrary name to a present state variable, and to every present state variable s we associate a next state variable NEXT_s.

For example, using the binary present state variable s, we may specify the transition relation of the automaton in Figure 3.2 as follows:

```
if s=0 then
    if x=0 ∧ y=0 then
        NEXT_s=0;
    elseif x = 1 then
        NEXT_s=1;
    else
        norun;
elseif s=1 then
    if x=1 then
        NEXT_s=1;
    else
        norun;
```

One can easily see the correspondence with the figure. To be complete, one would need to add initialization to the start of the description, and the whole thing need to be enclosed in a loop statement to realize the infinite behavior. Lastly, constructs to load next state values to present state variables need to be added.

The same description can be written more compactly by reordering:

```
if x=1 then
    NEXT_s=1;
else
    if s=0 then
        norun;
    if y=0 then
        norun;
    NEXT_s=0;
endif
```

3.3.1.2 Signals

Let X denote some non-empty and finite set of *signals* (with an implied attached event), and let the function R be such that it assigns to every signal $x \in X$ a finite and

non-empty interval of integers. We say that $R(x)$ is the *range* of x, and it represents the set of values that x can have.

We define signals, associate to each a set of (data) values, and define event assignment and value assignment functions to represent the (event, data) pairs for the signals at a particular time. Any partial function $f : X \mapsto \{0, 1\}$ is called an *event assignment.* Any partial function f assigning integers to signals, such that $f(x) \in R(x)$ for all x for which f is defined, is called a *value assignment.* If f is some event or value assignment function, then we use $\mathcal{D}(f)$ to denote the subset of X for which f is defined.

3.3.1.3 Transitions

A *transition* is a 5-tuple (p, v, s, o, q), where p is an event assignment, v and o are value assignments called *input value assignment* and *output value assignment*, respectively, and states s and q are the *present state* and *next state*, respectively. If either p or v are defined for some signal $x \in X$, we say that x is an *input signal* of the transition (p, v, s, o, q). If o is defined for x, we say that x is an *output signal* of (p, v, s, o, q). If $p(x) = 1$ for at least one $x \in X$, we say that p is an *input stimulus.* If p is not an input stimulus, we say that the transition is *spontaneous.*

Intuitively, p checks for presence (if $p(x) = 1$) or absence (if $p(x) = 0$) of certain signal events in the captured input assignment, and v checks signal values. If both of these are satisfied and a CFSM is in state s, then it can move to state q and emit signals in $\mathcal{D}(o)$ with values specified by o. Note that there is no separate output event assignment. It is implicit that a signal is emitted if o is defined for it.

3.3.1.4 CFSM

Formally, a CFSM is specified as a FSM(see Section 3.1.2) with quintuple (I, O, X, R, F). We require that the present state of every spontaneous transition in F be an initial state, and that it has no incoming transitions. This implies that no reachable state in the machine has a spontaneous outgoing transition. Recall that CFSMs react to events, and hence are not allowed to execute spontaneously after initialization. The semantics of such spontaneity are undefined and hence disallowed.

We say that a signal $x \in X$ is an input (output) of a CFSM (I, O, X, R, F) if and only if it is an input (output) of at least one transition in F.

3.3.2 Semantics of CFSM

We define the behavior of a single CFSM in terms of the language of the associated automaton. The behavior of a CFSM network is defined by the composition of the associated automaton. The structure of the automaton and its constituent components associated with a CFSM is shown in Figure 3.3. Here, x and z are inputs, and y and w outputs.

The alphabet variables of the automaton associated with a single CFSM are:

- for every input x: Boolean variables `*x`, `read_x`, `read_*x`, and `*x_read`, and $R(x)$-valued variables `x` and `x_read`,
- for every output y: Boolean variables `*y`, `*y_sent` and `send_y`, and a $R(y)$-valued variable `y`,
- a Boolean variable `t`.

In addition, each component has some internal variables.

The associated automaton of a single CFSM is a composition of the following components:

- an *input buffer automaton* IB_x for every input x,
- an *output buffer automaton* OB_y for every output y,
- a *main automaton* M.

Partitioning and modularizing the mathematical definition in this fashion enhance the ease of understanding for this very complex model. The complexity of the model is a direct reflection of the complexity of embedded systems. Modularizing in this fashion also make for more efficient implicit representation [TSL+90]. Ordering BDD for a set of FSMs with regular structures is easier than a single, large FSM.

Roughly speaking, input buffer automata read signal events (e.g. `*x`) and data (e.g. `x`), while output buffer automata non-deterministically (but still following certain rules) emit the output signals. At the transition point of the CFSM (marked by `t=1`), the main automaton ensures that the signals emitted by the output buffer automata are consistent (with respect to the transition relation) with the signals observed by the input buffer automata. This allows the implementation full freedom to choose when to read and

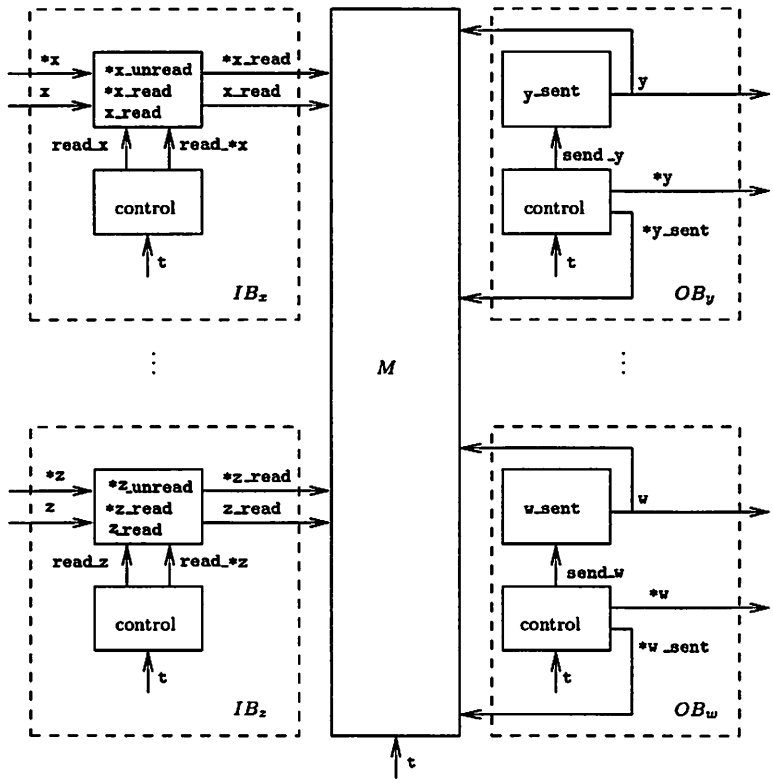

Figure 3.3: The Automaton Model of a CFSM.

emit signals between two transition points. A flexibility that can translate to efficiency at the implementation level.

In order to ensure that every non-deterministic emission is verified by the main automaton, we restrict the language only to those sequences in which every emission (i.e. `*x=1` for some signal x) is followed eventually by a transition point (i.e. `t=1`). Such a restriction is easily accomplished with *fairness constraints* [Tho90]). The behavior of a CFSM is the projection of the language of the associated automaton onto all the input and output signal variables `x` and `*x`. All the other variables are there just to help define the semantics, and need not appear in any implementation.

3.3.2.1 Input buffer automata

An input buffer automaton IB_x is the composition of a data part and a control part. Based on the read_x and read_*x commands from the control part, the data part reads the event and data of signal x, and stores them for the main automaton.

The control part is illustrated in Figure 3.4. It ensures that both the event and the value of the input will be read exactly once when t is set to 1. It also ensures that the event is read before the value. There are no restrictions on the timing of these operations, only that they respond to the input t.

The data part has three state variables. Two of them, *x_read and *x_unread, are Boolean, while x_read is $R(x)$-valued. A value of 1 for *x_unread means that an event came in, but the read signal read_*x from the control part has not come in (i.e., x has appeared but has not been read). A value of 1 for *x_read means that both an event came in, and the read signal came in; at this point the event can be communicated to the main automaton. Similarly, the x_read variable communicates the value to the main automaton. Note that *x_read and x_read are state variables for the data part of the input buffer automaton and also alphabet variables, since they must be communicated to the main automaton.

The initial value of *x_read and *x_unread must be 0, while x_read can initially take any value from the range of x. The transition relation of the data part is given by:

if read_*x=1 *then*
  if *x_unread=1 ∨ *x=1 *then*
    NEXT_*x_read=1;
  else
    NEXT_*x_read=0;
  endif
  NEXT_*x_unread=0;
else
  NEXT_*x_read=*x_read;
  if *x=1 *then*
    NEXT_*x_unread=1;
  else
    NEXT_*x_unread=*x_unread;
  endif

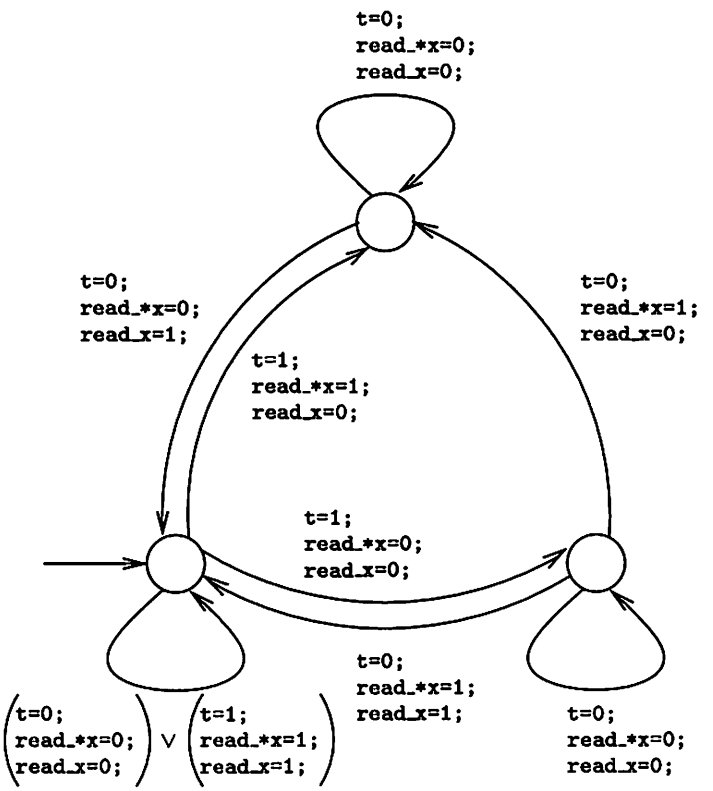

Figure 3.4: The Control Part of Input Buffer Automaton IB_x.

endif
if read_x=1 *then*
    NEXT_x_read=x;
else
    NEXT_x_read=x_read;
endif

Intuitively, the input event is latched in `*x_unread`. When `read_*x=1`, the current or latched value of the signal event is transferred to `*x_read`, and `*x_unread` is flushed (and ready to latch the next incoming value). Similarly, when `read_x=1`, the current value of signal data is transferred to `x_read`.

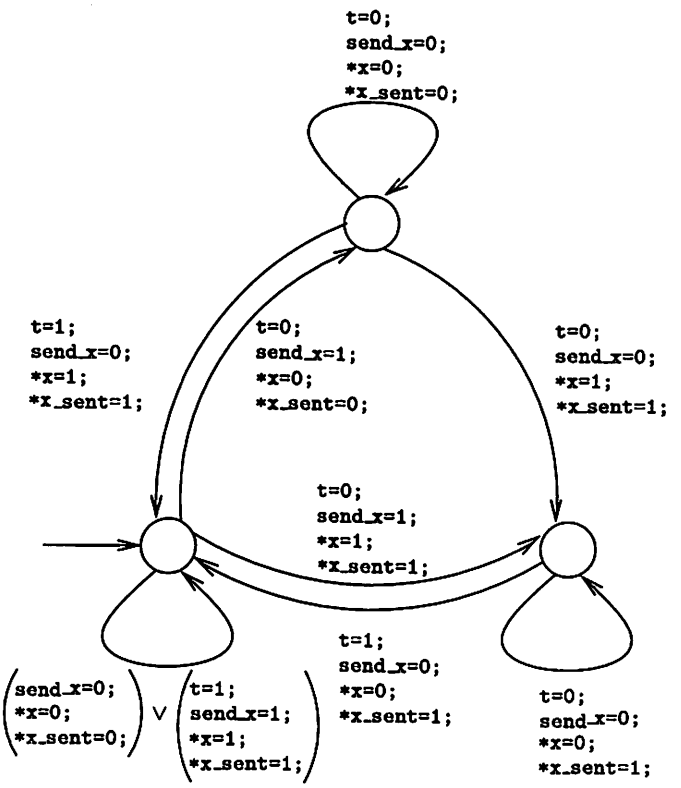

Figure 3.5: The Control Part of Output Buffer Automaton OB_x.

3.3.2.2 Output buffer automata

The data part of the output buffer automaton OB_x has a single $R(x)$-valued state variable x_sent, the initial state of which is arbitrary. The transition relation of the data part is given by;

x=NEXT_x_sent;
if send_x=0 *then*
    NEXT_x_sent=x_sent;
endif

Intuitively, when send_x=1, x takes an arbitrary value (determined by the main automaton to correctly implement the transition relation), which is then kept in x_sent until the next send_x=1.

The STG of the control part is shown in Figure 3.5. Between any two transition points it either generates both send_x=1 and *x=1, or neither. It will never generate *x=1 before send_x=1. If it generates *x=1, at the same time it also generates *x_sent=1, and keeps generating it until the next transition point. The main automaton will use the *x_sent signal effectively to set the output value.

3.3.2.3 Main automaton

The states and initial states of the main automaton M are the same as those of the CFSM (I, O, X, R, F). In the STG of M the edge $s \rightarrow q$ is labeled with (t=1)∧P if $s \neq q$, and ((t=1)∧P)∨(t=0) if $s = q$, where P is the following predicate:

$$\mathtt{P} \equiv \bigvee_{(p,v,s,o,q)\in F} \left(\bigwedge_{x\in\mathcal{D}(p)} \mathtt{*x\_read} = p(x) \right.$$
$$\wedge \bigwedge_{x\in\mathcal{D}(v)} \mathtt{x\_read} = v(x)$$
$$\wedge \bigwedge_{x\in\mathcal{D}(o)} (\mathtt{*x\_sent} = 1) \wedge (\mathtt{x} = o(x))$$
$$\left. \wedge \bigwedge_{x\notin\mathcal{D}(o)} \mathtt{*x\_sent} = 0 \right) .$$

Intuitively, when t=0 the state cannot change, and when t=1, no transition is enabled unless P is one, and that is true only if signals emitted at the outputs and the signals observed at the inputs are consistent with at least one transition in F.

3.3.2.4 Networks of CFSMs

A network of CFSMs is defined by simple composition of the automata representing individual CFSM. The transition variable t for each CFSM is completely free and all possible interleaving and concurrent executions of the CFSMs are represented and consistent with the CFSM network specification. An implementation is the refinement of the behavior of the transition variables according to architectural mapping and the synthesis directives for the implementation. For example, a Unit Delay Parallel (UDP) scheduling policy on some FPGA dictates that the transition variables t for all CFSMs be 1 at the same time, and the intervals between the adjacent occurrences are exactly equal to the

"clock" period. Static Priority Serial (SPS) scheduling policy on some processor has individual t to be mutually exclusive. The ordering of executions is set according to some statically determined priority. The timing between the transitions is dependent upon the delays of the executions, which in terms depend on the architectural mapping and synthesis directives.

Chapter 4

Formal Verification of CFSM Specifications

Design verification of embedded systems is traditionally performed by prototyping and simulation. Prototyping is clearly expensive in terms of turn-around time, and, in addition, cannot be performed until most of the detailed design is completed. Simulation is valuable, but for complex systems, only relatively few input patterns can be tried, thus reducing the power of simulation in exposing errors in design.

Formal verification is a set of techniques that facilitate proving mathematically that some formally specified properties are true for a design. These techniques are obviously very powerful, as they do not rely on the completeness of a set of simulation vectors. However, the computational complexity is very high. We believe that formal verification can be a useful tool for early error detection, especially for system design. At the early stage of the design process, a design is usually specified at the behavioral level. Fewer implementation details are present in the specification, so the overall complexity of analysis through formal verification is usually lower at this stage.

Formal verification requires a formal model of the behavior of a system, as well as of the properties we wish to verify. Figure 4.1 provides a general paradigm for formal verification. A system description, preferably a behavioral specification that is devoid of all unnecessary implementation details, is entered into the verification tool, along with a set of properties. If the verification tool determines that the system description satisfies those properties, we can then go to the synthesis stage. The synthesis stage may provide

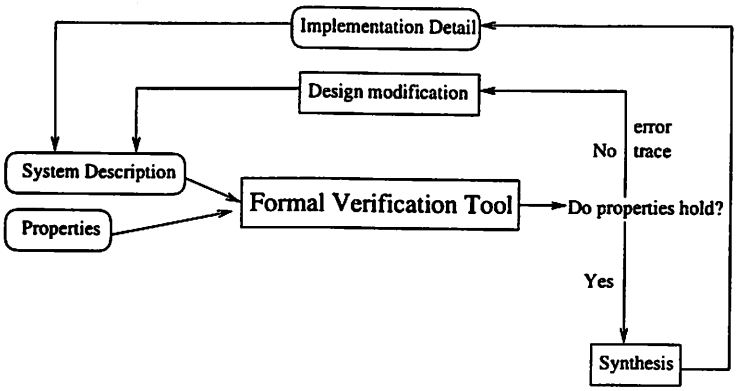

Figure 4.1: A Generic Formal Verification Paradigm.

some implementation detail, which may be necessary to verify some other properties that the designer has in mind. If the verification tool determines that the system description does not satisfy those properties, it will produce an error trace. This trace is a sequence of inputs and state transitions of the system, starting from an initial state. The designer can use the error trace to discover the source of the error and modify the design accordingly.

In this chapter, we show how an automata-theoretic approach to formal verification can be applied to CFSMs. In the automata theoretic approach [Kur94], systems are modeled by automaton and the language of the automaton (i.e., the set of sequences of inputs and outputs observable at the ports of the system) is taken to be the behavior of the system. The task of formal verification is to show that all these sequences are "acceptable". In one possible approach to formal verification [Kur94], acceptable sequences are specified as a language of another automaton, so the verification problem reduces to checking language containment between two automata. Another approach, called model checking ([BCL+94]), is based on labeling states of the system automaton with properties (described in some form of temporal logic) that they satisfy.

The main advantages of the automata-theoretic approach (including both language containment and model checking) are that it can be completely automated, and that it allows conservative abstractions to reduce the complexity of the computation. Let A be some automaton. If A is modified (say to A') in a manner that can only add new sequences, but never eliminate a sequence from the language, then A' is a conservative abstraction of A in the sense that A satisfies all the safety properties that A' does (i.e., the language of A is contained in the language of A', therefore, all the languages that contain the language

of A' also contain the language of A). Conservative abstraction for "liveness" property is more complex and will not be considered in this dissertation.

Practically, abstraction is applied to variables in the following two ways:

1. Free a variable completely. For the property that we are analyzing, it may not matter at all what the value of some variables may be. Hence, we can safely remove all the registers that are used to store these variables and all the components that are used to compute them.

2. "Reduce" the range of the variable. For the property that we are analyzing, it may only matter whether a variable is larger, equal to, or less than a constant, say 25. We can then replace its range from 2^{16} (if the underlying integer representation has 16 bits), to 3 values: < 25, $= 25$, and > 25. If we have 5 such variables (or 5 copies of such a variable), we can reduce the state space contribution due to these variables from over 10^{40} to less than 1000.

Often, even relatively simple systems are too complex to be handled by any formal verification tool existing today. It is not unusual for a designer to specify a design that has a tremendous number of states. For example, a single hardware circuit with 500 latches has more that 10^{100} states, and a single program with 10 32-bit integer variables has more than 10^{80} states. Even the best existing formal verification tools can routinely verify systems with no more than 10^{30} states (even though some larger systems can be verified if they have a special structure). Thus, some abstraction must always be applied to enable verification.

Figuring out automatically what abstractions should be applied for a given property is computationally just as hard as verifying the property without those abstractions. Hence, abstractions must be chosen by the user, but they can then be propagated automatically. In addition, if an abstraction chosen from the two classes outlined above happens to be "incorrect", so that it affects the outcome of the verification, then it must be *conservative*. This means that it can only turn a positive result into a negative result (a "false negative"), and never turn a negative result into a positive one (a "false positive"). If the result of verification is that the property does not hold for the abstracted system, the user can analyze the source of the error through an error trace supplied by the verification tool to see whether it is due to abstraction or a true error in the design.

Neither CFSMs nor automata deal with quantitative timing issues. In fact, CFSMs are defined with the assumption that a reaction to an event can take an un-

bounded amount of time. This assumption provides an implementation-unbiased starting point for a verification-driven design methodology. The first step in that methodology is to try to verify the system with unbounded delays of the CFSMs. If the verification fails (as it most often will), the error trace is analyzed in order to suggest timing constraints necessary to satisfy the property. Then, the verification is attempted under the assumption that timing constraints are met. If successful, the used assumptions are recorded as constraints to be met by the implementation. In this way, a verification tool is used as design aid to refine the specification of the system. An example of this type of assumption is one that can be placed on the correct scheduling. If the designer discovers through the error trace analysis that overflow on a certain input buffer leads to an error condition, then an assumption on the correct scheduling (i.e., that a specific input buffer never overflows) is placed on that input signal, and verification continues. If these constraints are consistent and implementable, downstream synthesis algorithms can then take this assumption into consideration and implement it correctly.

4.1 Verification Methodology

To be able to apply the automata-theoretic verification approach to CFSMs, we follow the procedure in Section 3.3 to generate for a given CFSM a set of automata that together represents the behavior. Practically, the implementation of this procedure is a translator from SHIFT to a format understood by verification tools. The verification tools we currently interface to are VIS [BSVA+96] and HSIS [SAB+93]. They both read in a description of the system and the property written in a format called BLIF-MV, build the internal representation in terms of Binary Decision Diagrams [BRB90], and then verify the system.

Verification of the FSMs translated from CFSMs uses the following five steps:

1. Verify general properties or error conditions.
   Verifying "normal" operation is equivalent to a restatement of the system functionality. Hence, it is generally much easier to verify general properties, like "absence of deadlock","mutual exclusion", or checks for the system response to particular conditions (often error conditions).

2. Decomposition.

Verify simple properties. Together with "back of the envelope" logic implication, the designer can prove complex properties. Alternatively, theorem proving ([ORR+96, GM92]) could be used to systematize this decomposition while automata-theoretic techniques are used to prove the sub-properties.

3. Localization.

   Verify local properties. Break down the property into smaller bits pertaining to only one or a few modules. Together with "back of the envelope" logic implication, the designer can prove the complete desired property.

4. Abstraction.

   Abstraction is the process of eliminating specific details that are not relevant to the property at hand. CFSM is a modular representation in which control and data computations are modeled separately. This makes abstraction easier to perform. To guarantee that abstraction is conservative, we limit our abstraction operations to either freeing the entire variable, or breaking up the range into equivalent regions. For each variable, the designer can determine the distinguishing values for the property at hand. For example, given a variable as output of a counter. The property may only care if the counter reached the counted value, say 25. Then the output of the counter has only two distinguishing values: ≤ 25 and > 25. In this case, we have two equivalent regions.

5. Assumptions on timing (scheduler).

   For some designs, it may be necessary to assume that CFSM scheduling is "correct", meaning that some CFSM input events are never lost, in order for some specific property to hold. This assume-guarantee paradigm is a system level extension to the one in [GL94], treating scheduler also as a component. There are two consequences of this assumption:

   - Separation of timing and functionality.

     Timed verification is generally so complex that it cannot be applied to most realistic design. However, we can argue about the correctness of a particular form of timed property (overflow of input buffer) by using only untimed verification tools. The complementary process, of verifying timing by using functional assumptions, is still a topic of active research [Bal99].

- Verification-directed architectural mapping.

  Most verification tools provide an error trace for a failed verification run. The designer can analyze this error trace. If the error is due to a buffer overflow, and if an assumption on the correct timing can eliminate this error, a downstream design constraint is placed on that variable and that CFSM. This particular type of constraint may be satisfied by the scheduling policy, or by assigning the CFSM to a hardware implementation, or by introducing a more complex handshaking mechanism at the specification level.

4.2 Verification Example: Seat Belt Alarm Controller

We first present a simple verification example, the seat belt alarm controller discussed in Section 2.1. The timer is assumed to be part of the external environment. While too small to demonstrate the power of abstraction, or the need to make assumptions on the scheduler, it is easily understood. We will present a real-life verification example in the next section.

The question we would like to answer through formal verification is: "Can the alarm sounds continuously?". It would be most annoying if some specification bug or possible combination of design choices could yield a product that reacts to some input sequence by leaving the alarm on forever. More formally, the property we want to verify is:

P1: The alarm eventually goes off.

If the seat belt alarm controller ever turns the alarm on, then it must turn the alarm off some time in the future.

This property is a general property of the seat belt alarm controller because it is a general statement about the "goodness" of the system, not a detailed statement about one particular execution of the system (as in simulation). It is also local to the seat belt alarm controller, in that only moderate assumptions on the environment are needed in order to prove it.

After translating the CFSM into a network of FSMs (or equivalent finite automata) according to the procedure outlined in Section 3.3, we checked property P1 by writing it as a monitor which interacts with the network of FSMs. This monitor is shown as an FSM in Figure 4.2. The monitor stays in state "O.K." until it sees alarm_on, then it

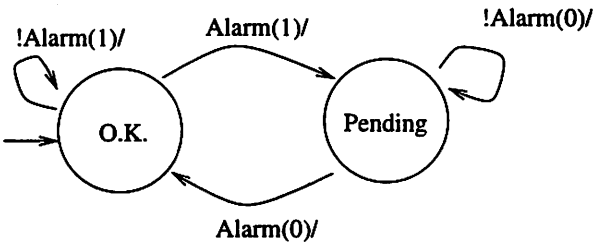

Figure 4.2: FSM Monitor for Seat Belt Alarm Controller.

transitions to state "Pending". It stays in "Pending" till it sees alarm_off, then it transitions back to state "O.K.". We want to see if it is possible that the monitor remains in state "pending" forever. Verification will be accomplished by the used of fairness constraints. Description of fairness constraints and the details of the operation of verification tools are beyond the scope of this dissertation. Please see [BSVA+96] or other verification documentation. The notation "!" is used to denote the absence of the event. Note that unlike CFSMs (which are a reactive model and respond only when there is at least one present event), FSMs and automata may make a transition even without any present event.

The initial run of the verification tool gives us an error trace. By looking at the error trace, we realize that we must further limit the behavior of the environment, namely the generation of Key_On, Key_Off, End_5, and End_10, which up to this point is considered completely non-deterministic. It is fair to assume that the environment can only generate Key_on, and Key_off alternatively. This is captured in FSM form in Figure 4.3. In state 0, the FSM can non-deterministically choose to stay in state 0 or go to state 1 and emit Key_On. In state 1, the FSM can non-deterministically choose to stay in state 1 or go to state 0 and emit Key_Off. By not specifying any precondition on a transition, we are saying that it can transition on anything at the next cycle.

Another reasonable assumption on the environment is the abstract behavior of the timer, represented in Figure 4.4. The FSM waits for a Start signal, then it goes to state 1. It could stay at state 1 non-deterministically, or it could go to state 2 and emit End_5 as long as another Start is not detected. If it receives a Start at state 2, it is reset and returns to state 1, otherwise, it can choose to stay at state 2, or emit End_10 and go back to state 0. This timer FSM is really an abstraction of what would be translated from the Esterel description of the timer in Section 2.1, because it is simpler and emits End_5 and End_10 in sequence after Start, but not necessarily after 5 and 10 seconds. A fairness constraint is

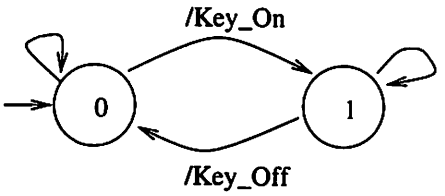

Figure 4.3: FSM Representation of the Behavior of the Key Signals.

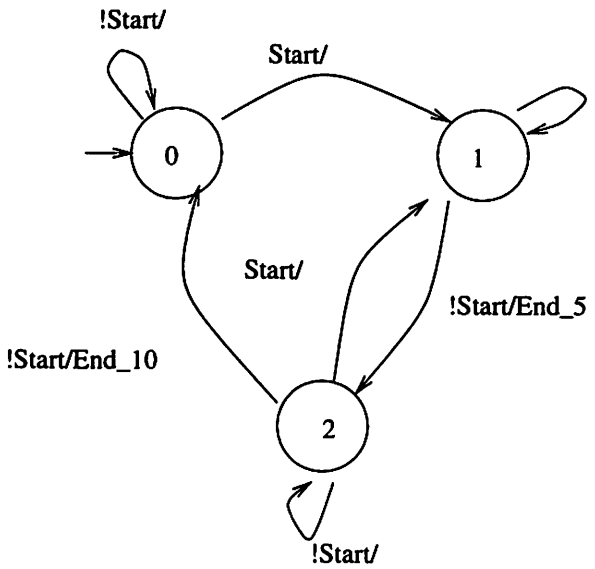

Figure 4.4: FSM Representation of the Abstract Behavior of the Timer.

put on this FSM such that it eventually goes back to state 0, i.e., it will not count forever. (This is not part of the automata, but a characteristic given to it, and that it is assumed to have in the implementation.)

With these specifications of the environment, property P1 still does not hold. One can find the problem by looking at the error trace. The original CFSM in Figure 2.4 does not specify what happens if End_5 and End_10 are both present in some CFSM execution. In a particular implementation, the CFSM can be so slow that it does not react to its input in 5 seconds. Hence, it can see both End_5 and End_10 at its input buffer at the same time. The Esterel file in Section 2.1 actually gives End_5 a higher priority than End_10. This means that the alarm will stay on forever in this case. A similar problem can occur also with Key_On and Key_Off, as well as with Key_On and Belt_On (albeit under different conditions). A revised Esterel specification that passes property P1 is:

```
module controller:
input Reset, Key_on, Key_Off, Belt_On, End_5, End_10;
output Alarm(boolean), Start;
loop
  do
    emit Alarm(false);
    loop
      do
        every Key_On do
            emit Start;
            do
              await End_5;
              emit Alarm(true);
              await End_10;
              emit Alarm(false);
            watching End_10;
            emit Alarm(false);
        end
      watching [Key_Off or Belt_On];
      emit Alarm(false);
    end
  watching Reset
end.
```

There are two major modifications from the previous version of the seat belt alarm controller. The first is in the `do ... watching End_10`. This is to make sure that if `End_5` and `End_10` occur at the same time, `Alarm(false)` will still be emitted. The second is in putting `do ... watching [Key_Off or Belt_On]` outside of `every Key_On do ... end` instead of inside it. This is so that if `Key_Off` or `Belt_On` is read together with `Key_On`, `Alarm(false)` will still be emitted to turn off the alarm.

4.3 Verification Example: Shock Absorber Controller

In this section we present the verification of a shock absorber controller. The design comes from a realistic specification of a commercial product of a major European automotive subsystem supplier. The existing hand-designed board contains a single Motorola 68HC11E9 microcontroller in expanded mode, with 32 Kbytes of external EPROM and 8Kbytes of external RAM, plus on-chip RAM and EPROM. The design effort required is approximately one engineer-year. The natural language specification is coded as 4507

lines of Esterel statements divided into 29 modules.

The controller sets the shock absorber motors to a HARD, MEDIUM, or SOFT level, according to values from a set of sensors: steering wheel, vertical acceleration, speed, and battery voltage. Figure 4.5 shows the structure of the complete controller. The modules LONG_SPEED, LONG_ACC, VER_ACC, STEER_ANG, and STEER_SPEED compute the horizontal speed, horizontal acceleration, vertical acceleration, steering angle and steering speed respectively, from the input sensors, and suggest the shock absorber level based on the appropriate look-up table. The module BAT_DIAG sends out signals if the battery voltage is out of the given range. The module MOT_CTRL records current suggestions of all the modules and sets the motors to the hardest of suggested values.

During normal operation, the output, MOT_CTRL, is continuously updated according to input sensors. At the same time, input modules continuously check input sensors for erroneous conditions, and send a signal if such a condition is detected. The specification requires seven different conditions to be checked, and appropriate actions to be taken if errors are detected.

P1: Speed parasitic(glitch): If the speed sensor indicates impossibly high speed on more than three occasions, the shock absorbers are to be set to HARD until the RESET event occurs.

P2: Open circuit: If there is an open circuit condition on the speed sensor, the shock absorbers are to be set to HARD until RESET.

P3: Battery: If the battery voltage is out of range, the shock absorbers should be set to HARD. If this condition persists for some time, then the setting should remain HARD until the RESET event occurs.

P4: Speed sensor: If the horizontal speed appears to be zero, and shaking (vertical acceleration) is detected, then the shock absorbers are to be set to HARD until the RESET event occurs.

P5: Steering speed: If the steering speed is too high, the shock absorbers should not be set to SOFT for at least the following minute. If this condition occurs more than three times, the shock absorbers are to be set to HARD until the RESET event occurs.

P6: Vertical acceleration: If the number of times the vertical acceleration sensor reports

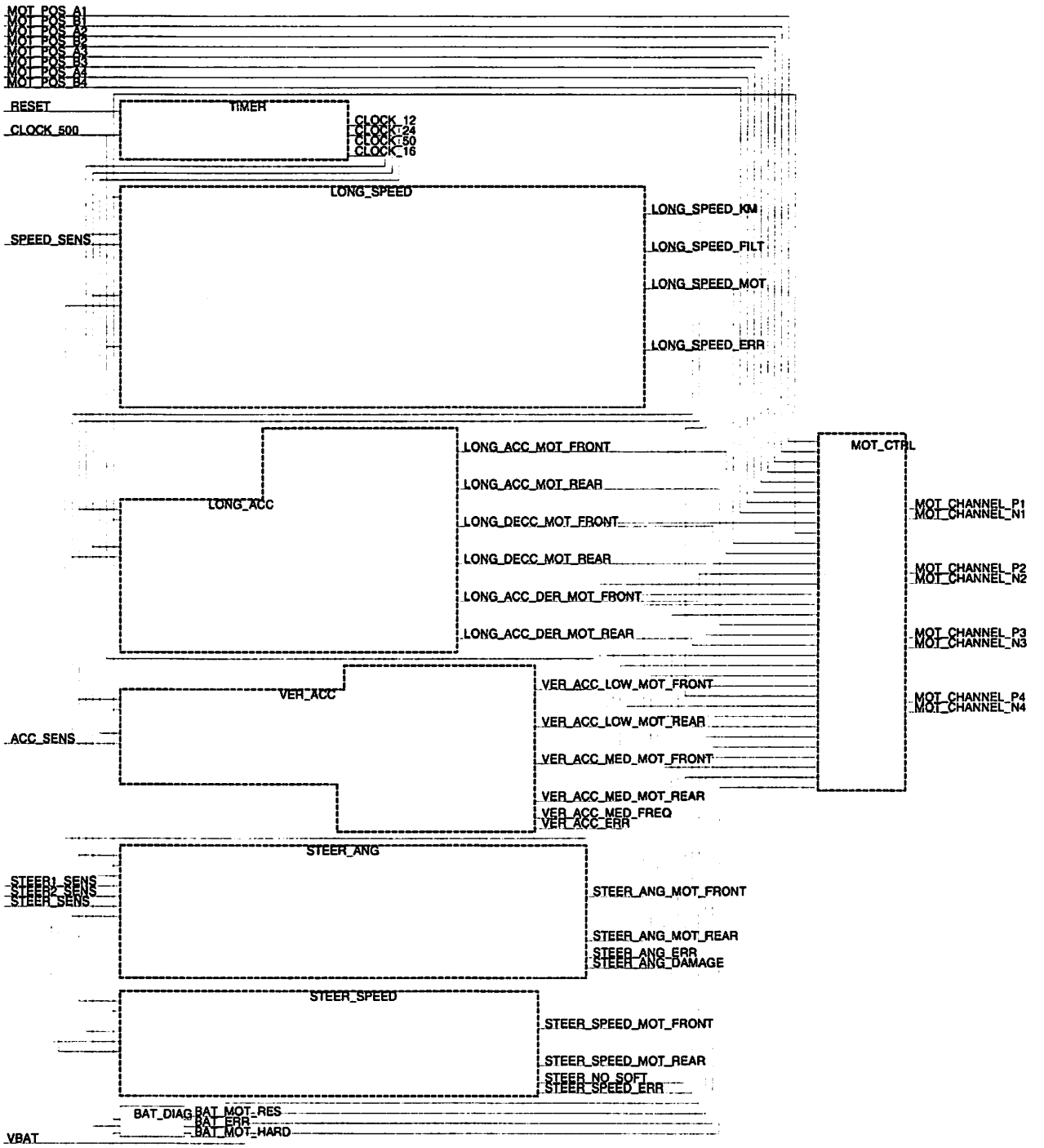

Figure 4.5: The Shock Absorber Controller.

out-of-range data exceeds some given limit, then the shock absorbers are to be set to HARD until RESET.

P7: Steering angle: There are two steering sensors. If there is no change in the sensed values for either of the sensor for 1 minute, the shock absorbers are to be set to MEDIUM. If this condition persists for 4 minutes, the setting should be HARD. If it persists for more than 12 minutes, then the setting should remain HARD until RESET, even if some changes are later detected. Similarly, if there is no change in the sensed values for one of the sensor for 2 minutes, the setting should be MEDIUM. After 8 minutes without a change, the setting should be HARD. After 24 minutes, the setting should remain HARD until RESET.

The system responds correctly to error conditions if and only if the above seven properties hold. Here we present in detail verification only of the first one. Since all the properties have a similar form, the other properties are verified in a similar fashion. Verifying one property at a time enables us to abstract signals and modules not related to the property at hand.

The intended behavior of the system relevant to the property **P1** is as follows. The error is first detected by the module called LONG_SPEED_DIAG_PAR (see Figure 4.6). To understand the behavior of that module more precisely, we first need to explain how speed is calculated. The speed sensor generates a sequence of events (called SPEED_SENS), the frequency of which is proportional to the speed of the vehicle. Speed is calculated by counting clock events (called CLOCK_500) between any two occurrences of SPEED_SENS events. The count is stored in variable D_TIME. Even at the highest possible speed of the vehicle, there can be no less than 6000 CLOCK_500 events between two SPEED_SENS events. Thus, if D_TIME is less than 6000 when the new SPEED_SENS event occurs, that must be a parasitic signal (glitch). Upon detection of a glitch, no immediate action is taken. However, the number of detected glitches is recorded (in variable MIN_TPAR_NUM), and if that number is larger or equal to 3 during any period of CLOCK_500, then an error event (called ERR_PAR) is emitted.

The ERR_PAR event is received by a module called MOT_CTRL_DAMAGE. It records error events from all the input modules (including SPEED_DIAG_PAR), and generates the DAMAGE event with a value HARD, MEDIUM, or SOFT, as required by the specification (see Figure 4.7). This error status is stored and can only be cleared upon

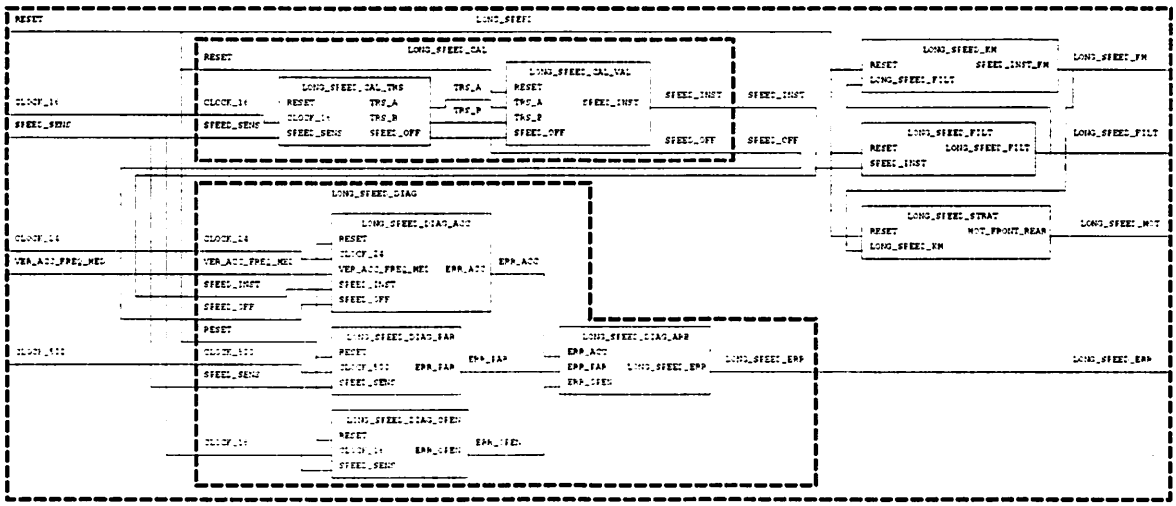

Figure 4.6: The Longitudinal Speed Strategy.

RESET.

The DAMAGE event is received by a module called DRIVER. It also receives another event called COMMAND_IN (the suggested setting of motors computed from look-up tables used in normal operation), stores them both, and generates the COMMAND_OUT event with the harder of two values.

The verification can be further simplified by decomposing a property into "local" sub-properties that each module must satisfy. In particular, we prove property **P1** by proving the following:

P1.1: If the speed sensors indicate impossibly high speed at least three times and no RESET events occur, then the ERR_PAR event will be generated.

P1.2: If the MOT_CTRL_DAMAGE module receives the ERR_PAR event, and no RESET events occur, then a DAMAGE event with a value HARD will be generated, and no DAMAGE event with some other value will be generated before the RESET event occurs.

P1.3: If the DRIVER module receives the DAMAGE event with value HARD, then the setting of the shock absorber will be HARD, and the setting will not change until either a RESET event, or a DAMAGE event with a value other than HARD occurs.

Often, properties decompose naturally into local sub-properties, and a simple check

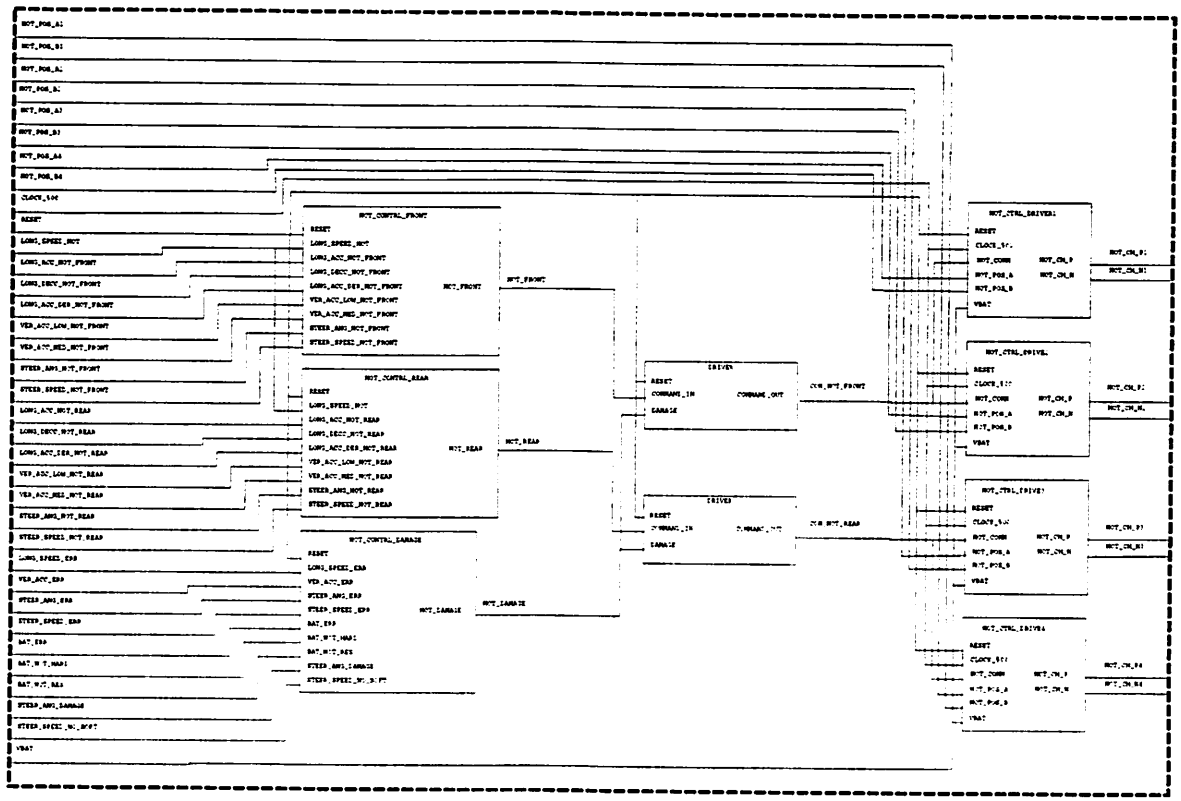

Figure 4.7: Motor Control.

is needed to verify that sub-properties indeed imply the desired property. Finding a good, correct decomposition can be a hard task, and it cannot be completely and efficiently automated. In non-trivial cases, one needs to use another formal technique (e.g., automated theorem proving [ORR+96, GM92]), to show that a set of sub-properties implies a property.

4.3.1 Verifying Property P1.1.

The precise formulation of the error condition is as complex (and thus as error-prone) as the description of the module. To avoid the problem of incorrect property specification, we verify the following simple property instead:

P1.1': If four SPEED_SENS events occur between two CLOCK_500 events, then the ERR_PAR event is generated.

This obviously covers only a small portion of the intended behavior, because any other "impossibly high speed" will not be captured by this property. However, it is often the case

that such simple "sanity checks" reveal interesting bugs in the system.

The automaton obtained from the original CFSM specification by the procedure in Section 3.3 has 141 latches. The formal verification tool HSIS [SAB+93] and VIS [BSVA+96] both run out of the 480Mb of main memory before even constructing the internal representation.

4.3.1.1 Verification with Abstracted Integers

The CFSM, on closer inspection, consists mainly of two 16-bit integers used as counters, and two comparators comparing these integers to constants. However, for the property we are interested in, only one value of D_TIME is distinguished: 0, indicating that no CLOCK_500 events have occurred between two SPEED_SENS events. We can thus abstract this counter to only two states: "0" and "> 0", and modify the comparator and incrementor accordingly. If the input of the incrementor is 0, the output must be > 0, and if the input is > 0 the output is chosen non-deterministically to be > 0 or 0 (due to possible overflow). Similarly, the comparator checking whether D_TIME is less than 6000 is modified so that if its input is 0, the output is 1. If its input is > 0, then the output is either 0 or 1. In a similar way, we abstract MIN_TPAR_NUM to four distinguished values: 0, 1, 2, and ≥ 3. These abstractions introduce some new behaviors (e.g., when the actual value of D_TIME is larger than 6000 but the comparator still outputs 1), but these additions do not affect the outcome of verification. It is important to notice that these abstractions can be done automatically, by syntactic modifications of the BLIF-MV description of the system, so that the size of the description is guaranteed not to increase. However, deciding which abstractions are consistent with the property is as hard as the original verification problem, so user guidance is crucial in this step.

With these abstractions, building and verifying the model takes less than 10 seconds. Unfortunately, the property fails. The error trace indicates that the CFSM can react too slowly, so SPEED_SENS events can be over-written without being sensed by the CFSM (e.g., if it is not assigned a high enough priority by the scheduler).

4.3.1.2 Verification Under Timing Assumptions

Indeed, the property can be satisfied only if some timing assumptions are made about the CFSM. We will follow the usual methodology of separating timing and functional

properties as much as possible. More precisely, we will verify the conditional property: "if the CFSM reacts to every SPEED_SENS event, then the property holds". Such a property is local, i.e., it depends only on the behavior of the CFSM. On the other hand, verifying that indeed the CFSM reacts to every SPEED_SENS event is a separate problem that need to be addressed separately. We will just note that it is a hard problem, because it involves the characteristics of the hardware, of all the other tasks in the system, as well as those of the scheduling algorithm.

Fortunately, there is a very simple way to restrict the behavior of the system to the case where no input events are ever over-written. It suffices to remove from the transition relation of the input buffers IB_i all the elements that correspond to overwriting. Again, this modification requires a simple syntactic change on the intermediate description, which does not affect its size.

The conditional property showed an error in the design. Under the original specification, the property is not satisfied if the error condition occurs immediately after initialization. After correcting this error on the specification, the conditional property is verified in less than 10 seconds of CPU time.

4.3.2 Verifying Property P1.2

The automaton modeling MOT_CTRL_DAMAGE has 81 binary latches. The tool HSIS [SAB+93] could not construct the internal representation of the module even after several hours of computation time. It is expected that even with the drastic improvements in the implementation of the verification technology, the huge state space of the example also prohibits direct use of VIS [BSVA+96] in verification.

4.3.2.1 Using Timing Assumptions and Freeing Variables

For the property at hand, the values of all the internal variables, except the one holding the value of ERR_PAR, are irrelevant. We can thus "free" them, i.e., allow them in every execution step to take any value from their respective domains, and remove from the model all computation of these variables. Again such an abstraction can be done automatically with a single pass through the intermediate description of the system, but deciding which variables can be freed is generally a hard problem.

With this abstraction, the verification time is reduced to a few seconds. The

property is verified under the assumption that the system cannot ignore an input event forever. Note that this is a weaker assumption than the one requiring that every input event be detected, because it still allows "over-writing" of input events. The verification of the conditional property takes less than a minute of CPU time.

4.3.3 Verifying Property P1.3

The module DRIVER is simple enough that it can be verified without any abstractions. During the verification, an error was found: the DAMAGE event was ignored if the COMMAND_IN event occurred simultaneously with it. After correcting the description of the DRIVER module, the property was successfully verified.

4.4 Conclusions

We have shown how existing formal verification tools can be used to verify properties of CFSMs. We draw two conclusions from this experience:

- Reactive systems of interest often have state spaces that are too large to handle for existing formal verification tools, thus the use of abstraction is crucial. Typical abstractions can often be performed automatically, by syntactic modification of the intermediate representation of automata. However, choosing the abstraction that is appropriate for the property to be verified can only be done by a designer with knowledge of the intended behavior of the system.

- Even though CFSMs are based on the unbounded delay assumption, most of the properties can be verified only if some timing constraints are imposed on the behavior of CFSMs. The assumption of the CFSM reacting to *every* input events provides a convenient way to separate timing and functional considerations. Verifying that this assumption is satisfied depends mostly on timing characteristic of other CFSMs in the system, as well as the environment and the scheduling algorithm. On the other hand, checking that the property is satisfied under this assumption, typically depends only on the functionality of the CFSM.

It is natural to ask, following the first conclusion, that for a specific property of a specific type of systems, whether there could exist an abstraction that works for all the

designs. The second conclusion leads us also to think that if the "separation of timing and functionality" is an inherent characteristic of the specification method, then one can exploit this characteristic to verify systems. For the rest of this dissertation, we concentrate on a subclass of specification where timing and functionality is strictly separated. We also restrict ourselves to a single property: the equivalence of two implementations. We develop a set of formal algorithms that are automatic and efficient. They can be used to find out whether two implementations are functionally equivalent to each other, or whether an implementation is functionally equivalent to a reference (golden) model.

Chapter 5

Synchronous Equivalence

Formal verification tools such as [McM93, BSVA+96] are very powerful in automatically proving that some arbitrary properties hold for an arbitrary design. Essentially, they perform exhaustive simulation for all possible input traces or system design parameters in an efficient manner. Unfortunately, for real-life embedded systems, the size of the design and the complexity of the analysis algorithms themselves cause the computer to consume too much time or memory. The verification is often unable to complete. To remedy this complexity problem, extensive user intervention is required to abstract away unnecessary details in the design representation. Figuring out the right abstraction for a given property is as hard as the verification problem itself, though designers usually have ideas as to what is the right abstraction for a given property. User intervention in the formal verification methodology is unavoidable.

The abstracted representation is not an exact representation of the design to be verified. In the context of formal verification as described in Chapter 4, it represents more behaviors than the exact representation. The same input trace can produce more than one output trace, due to the non-deterministic abstract representation. With these "added" behaviors, the result from the formal verification of safety properties [1] can be:

- Positive.

  Properties were satisfied.

- Inconclusive due to:

[1] These are properties of the type "something bad cannot happen". Within a real-time embedded context, liveness properties, such as "something good will eventually happen", automatically become safety properties, i.e. avoiding "bad" timeouts.

- True Negative.

  Properties were not satisfied, but they would have failed also for the exact representation.

- False Negative.

  Properties were not satisfied, though they would not have failed for the exact representation.

- Space Out or Time Out.

  The computation requires more time or memory than what is practically available.

It is impossible for the formal verification tools, by themselves, to distinguish between the true negatives and the false negatives. Obviously, no conclusion can be drawn when the computer spaces out or times out (and thus this is conservatively the same as a negative result). By examining the error trace, designers usually have ideas as to whether the inconclusive result is due to a true negative or a false negative. By examining intermediate results from the verification tools, designers can also identify portions of the representation that cause the computational problem. The representation can then be modified to exclude the false negatives or to abstract and simplify some portions of the system. User intervention, again, is unavoidable.

The fundamental difficulty in automatic verification of arbitrary properties of an arbitrary design is in finding the right abstraction level so that the verification can neither fail because of the false negatives nor because of the computational problems. It is desirable to have a property to either pass, or fail due to true negatives. This is in general impossible without user intervention. However, for a specific class of properties and a specific class of designs, it may be possible to establish simple formal algorithms at an abstraction level that are computationally efficient, and give inconclusive result that are *mostly* due to true negatives. By the choice of abstraction level, we hope to devise efficient algorithms that produce few inconclusive results that are due to false negatives or limited resources. The term "conservativeness" is used to qualitatively describe the amount of inconclusive results that are not due to true negatives. We want to come up with efficient algorithms that are not too conservative.

The specific property that we choose to investigate is one that determines whether two implementations of the same CFSM specification are *synchronously equivalent* to each

other. The class of designs for which we devised simple verification algorithms contains all the designs that satisfy *synchronous assumption*. Equivalence analysis is indispensable for verifying that some design optimizations did not alter the behavior or whether a physical implementation is consistent with a simulation model. In the next section, we provide further motivations for this choice of property and assumption, and in Section 5.2, we formally define them. In Section 5.3, we present a methodology for design exploration based on synchronous equivalence and synchronous assumption. In Section 5.4, we pave the way for establishing a set of analysis algorithms, trading-off computational efficiency with conservativeness. These algorithms will be presented in Chapters 6, 7, and 8.

5.1 Motivation

As we argued in the first chapter, a fundamental clarification to improve the design methodology is the formal definition of correctness. We advocate the principle of "separation of concerns" in verification. Functional correctness and timing are verified independently. This principle is the basis of the synchronous design methodology for sequential circuits [Ung69], where latches decompose the circuit into combinational islands. Signals are propagated from island to island when an enabling input (clock) is given to the latches. Any design of the combinational islands ensuring that the combinational circuits stabilize before the enabling signal arrives at the latches, can be verified for equivalence paying attention only to the Boolean functions computed by the circuits irrespective of the propagation time. Timing can then be verified independently by performing a worst-case timing analysis and making sure that this bound is within the clock cycle.

This powerful approach can be extended to higher levels of abstraction and to software design, as demonstrated by synchronous languages [BCG91]. Synchronous languages describe complex systems consisting of interconnected components each represented by a Finite State Machine model. Both communication and execution take zero time to perform. In practice, this means that the interaction with the environment has to be "much slower" than all the communication and execution time required for the reaction to the environment. In addition, the zero communication and execution times of the components imply unique execution order, if it exists, which depends entirely on the causal relationships among the component executions and not on the actual delays of the executions. While they are very powerful, synchronous languages support a model of computation that restricts the design

space considerably because of the synchronous communication hypothesis.

We relax the "synchronous hypothesis" by adopting a more general model of computation (the one supported by Codesign Finite State Machines, as described in Chapter 3), while retaining the fundamental idea of separation between timing and functionality. This is analogous to relaxing the synchronous assumption to the fundamental mode assumption when moving from synchronous to asynchronous circuits. We establish *synchronous equivalence*, a "functional" equivalence among a set of candidate implementations of embedded system specifications. Just as the sequential circuit design methodology abstracts away different gate delays among different implementations and enables speed/area trade-offs among functionally equivalent implementations, we abstract away the delays of embedded system computational resources. Synchronous equivalence lends itself nicely to simple analysis procedures so that we can figure out quickly whether two implementations are synchronously equivalent. The synchronous equivalence relation divides the design space into synchronous equivalence classes. Within an equivalence class, different implementations represent different trade-offs among speed, power, reliability, expandability, and cost.

Equivalence analysis can be done precisely through exhaustive simulation and reachable state analysis methods (e.g. formal verification tools [McM93, BSVA+96]), or conservatively (but more efficiently) through methods that are abstract, static, and structural. In Chapters 6 and 7, we derive efficient structural algorithms for synchronous equivalence analysis that can be used to explore the design space effectively. In Chapter 8, we relate these structural algorithms to precise, exhaustive simulation.

5.2 The Synchronous Assumption and Synchronous Equivalence

A key assumption is made on the class of "acceptable" specifications in order to make it easier to specify desirable behaviors and at the same time, make efficient analysis algorithms possible. We believe that this class includes many interesting examples, and that other specifications may be re-specified to satisfy this assumption.

Many popular design methodologies separate time into intervals of interaction with the environment and computation within the design. Software programs accept some input to start the computations. Under most circumstances, they finish computations be-

fore accepting new sets of input. Synchronous sequential circuits are hazard free and race free because interaction (propagation of state changes and signal changes) and computation (calculation of state changes and signal changes) are strictly separated. More recently, synchronous languages [BCG91] used the zero delay assumption, i.e. assumed that computation is always faster than the environment changes. It is therefore impossible for interaction and computation to overlap. We can thus imagine to strictly separate computation and interaction also for embedded system design based on CFSM. The advantage in ease of specification and analysis outweighs some sub-optimality due to the restriction in design style.

Definition 5.1 (Synchronous Assumption) *The operation of the design is split into two alternating* **non-overlapping phases.** *An interaction phase where the environment interacts with the design and a computation phase where the components in the design perform executions and communicate among themselves.*

This notion of synchronicity is a system-level extension to the fundamental mode operation of asynchronous circuits [Ung69]. During an interaction phase, the design "stands still" until the environment completes its receiving of outputs from and writing of inputs to the design for use during the next computation phase. During a computation phase, the design takes inputs that were generated by the environment, performs computations, and generates outputs that will be read by the environment during the next interaction phase. There are no interactions during a computation phase and no computations during an interaction phase. The interaction phase followed by its associated computation phase is called a "cycle". We will only consider specifications that satisfy this synchronous assumption. The implementation process must guarantee to preserve it. This can be done by a separate worst case timing analysis in the flavor of [BSV97]. If the worst case timing delay is within the constraints imposed by the environment, the implementation can be said to conform to the synchronous assumption.

Definition 5.2 (Synchronous Equivalence) *Under the synchronous assumption, two implementations are synchronously equivalent if and only if for all possible input traces the outputs of the implementations are the same at the end of every cycle.*

Note how internal events (among CFSMs) play absolutely no role in the *definition* of synchronous equivalence. As we will see, they may or may not play a role in *deciding* it.

As long as the results (outputs of the network of CFSMs) are the same at the end of the cycle, the order of executions of CFSMs or even the parallel/serial nature of the executions does not matter. The former can lead to freedom in scheduler selections while the latter can lead to freedom in assigning components to computational resources.

Our restricted CFSM model can thus be defined as externally synchronous globally asynchronous locally synchronous (ESGALS), as opposed to the original GALS CFSM model. The concepts of synchronous assumption and synchronous equivalence facilitate specification. Designers can now think about the input/output reaction of the design to the environment separately from the speed of the reaction. Synchronous assumption is not too restrictive, especially for control-dominated applications. Designing with the synchronous assumption is strongly analogous to designing synchronous circuits and fundamental mode asynchronous circuits. In that domain, the ease of validation and synthesis often outweighs the increased freedom with respect to full asynchrony. The same can be extended to embedded system design, board-level or "system on a chip", where different processors or dedicated units can be considered as different computational resources. Much research in embedded system design is also based on the synchronous assumption, as discussed below.

5.2.1 Related Work

Synchronous languages are a group of languages proposed for automatic synthesis of embedded software [BCG91]. Synchronous languages have a unique notion of "synchronous scheduler", the scheduler that defines correct behavior. This scheduler is the result of the assumption of synchronous communication *among components of the design.* Our synchronous assumption, on the other hand, is related only to the "external" interaction of the design with the environment. Hence, there is an intrinsic non-determinism in our specification that results in many possible "functional" behaviors that are consistent with the structural specification.

Synchronous data-flow is a powerful formalism for data-dominated embedded systems geared toward simulation and code synthesis for digital signal processors [LM87]. It also exploits the synchronous assumption at the interface between the network and the environment, but "blocking read" is required of all components in the design to ensure that the behavior is the same (in Kahn's sense [Kah74]) independent of allocation and scheduling.

Our work does not restrict the implementation choices to those utilizing a syn-

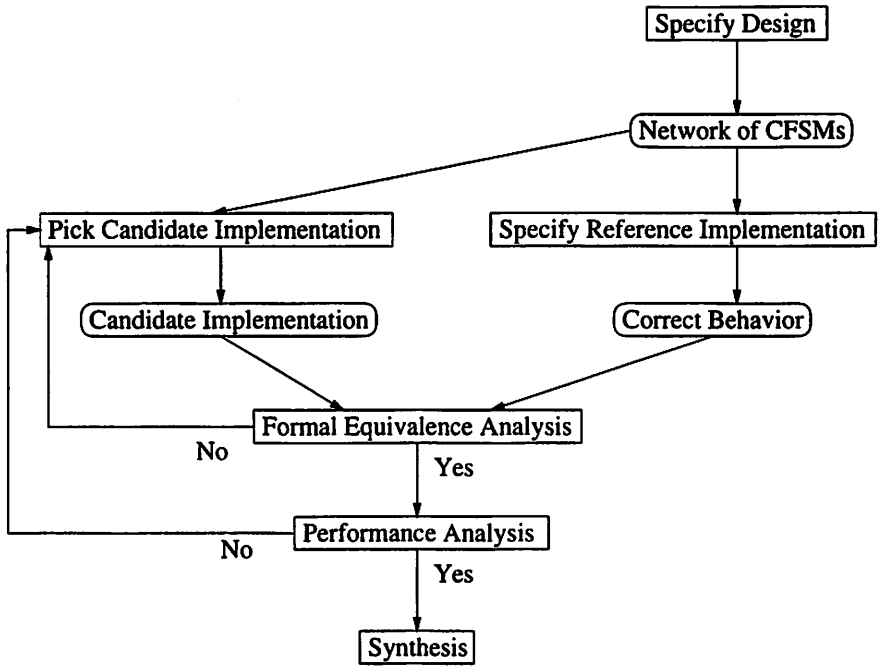

Figure 5.1: A Synchronous Approach to Design Exploration.

chronous scheduler, nor does it require communications to have the "blocking read" property. Therefore, many different functional behaviors are possible depending on the choice of implementation parameters. Different implementations are grouped into equivalence classes according to synchronous equivalence. We use equivalence analysis to tell us whether *any* two implementations are equivalent to each other. Trade-off analysis for different design metrics can then be performed on the "functionally" equivalent implementations.

Javatime [YMS+98] is inspired similarly by the fundamental mode operation of asynchronous circuits and the concept of refinement. Their emphasis is on the specification and simulation using the language Java. Our concentration, on the other hand, is on the development of efficient formal algorithms for equivalence analysis.

5.3 Design Exploration Methodology

Figure 5.1 illustrates a possible design methodology using the synchronous assumption and synchronous equivalence. The designer specifies the functionalities of the components and the structure of the design as a network of CFSMs. One or more be-

haviors among those allowed by the non-determinism in the network are chosen as golden model(s). They are represented by reference implementations that produce those functional behaviors and only those functional behaviors. The functional behaviors under synchronous assumption of these implementations are considered "correct" and different implementations with the same functional behavior are all functionally equivalent. Non-functional characteristics such as delay, power, reliability, and cost can be used to differentiate one functionally correct implementation from another. Separate analysis,in the flavor of [BSV97], of timing, power, or cost is performed to be make sure that constraints are satisfied and to decide whether one implementation is superior to others given some design metrics. The best one is then chosen to be synthesized.

5.4 Analyzing Synchronous Equivalence

Given any two embedded system implementations of the same CFSM specification, we want to determine whether or not they are synchronously equivalent. There are many possible approaches. One could exhaustively simulate through all possible input traces. Formal verification on exact representation can be a way to perform exhaustive simulation. On the other side of the spectrum, one may be able to deduce synchronous equivalence by statically analyzing the characteristics of the implementations. By static analysis, we mean that only simple comparison on implementation parameters or simple search algorithms on the system graph are allowed. In between these extremes are analyses where some abstract simulations are performed. The analysis techniques span a continuum on computation time and conservativeness of the analysis, as shown in Figure 5.2. Conservativeness is measured by the number of false negative results produced by the analysis algorithm, given a set of implementations of a CFSM specification.

Exhaustive analysis of any form produces no false negative results, but it often has very long computation time. All other methods have the possibility of false negative results given arbitrary designs. These methods, therefore, can only be used to prove that two implementations are synchronously equivalent. They cannot be used to prove that two implementations are **not** synchronously equivalent. This is a direct consequence of the "yes/inconclusive" nature of abstract analysis. On the other extreme from exhaustive simulation, one could therefore reach a conclusion of "don't know" without performing any analysis at all. In the design methodology proposed in the previous section, false negative

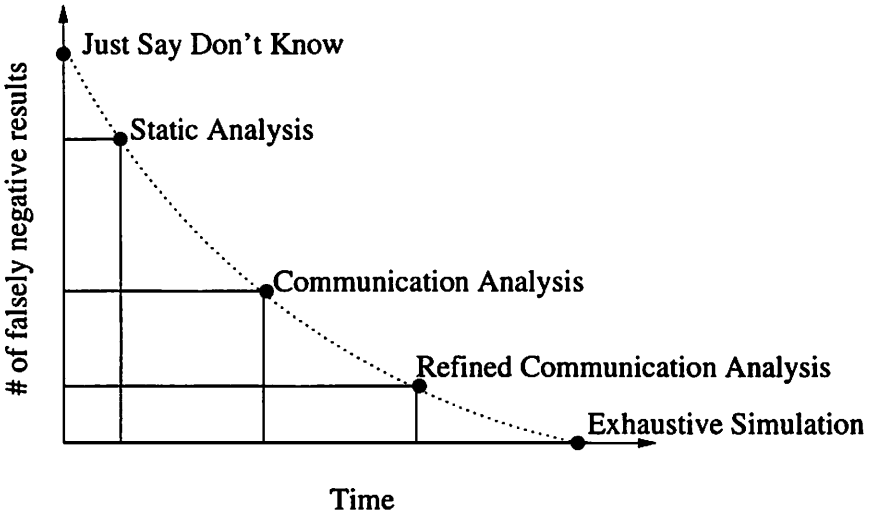

Figure 5.2: Trade-Off in Analysis Methods.

results will show up as candidate implementations that are falsely declared to be functionally incorrect. This may lead to sub-optimal final implementation. False positive results, on the other hand, may produce incorrect implementations and hence must be avoided. Static analysis methods can be very fast but generally produce many false negative results. In Chapter 6, we present simple static analysis algorithms that give useful results for CFSM network implementations, as supported by examples. In Chapter 7 and Chapter 8, we introduce Communication Analysis and Refined Communication Analysis that will produce fewer false negative results at a cost of higher computation time.

Chapter 6

Static Equivalence Analysis

In this chapter we present several static algorithms for the synchronous equivalence analysis of CFSM networks. Only simple comparison on implementation parameters or simple search on the system graph is needed for these algorithms. Consequently, they have extremely low bounds on maximum computation time, at the cost that many inconclusive results are possible due to the false negatives. We will show through examples that they are still very useful for embedded system design based on CFSM networks.

In the next section, we show how equivalence can be established by just examining implementation parameters such as scheduling policies. In Section 6.2, we show that some properties of the system graph can be used to guarantee synchronous equivalence. In Section 6.3, some analysis result based on both implementation parameters and properties of the system graph is presented. In Section 6.4 we show how some popular implementations of heterogeneous architectures are synchronously equivalent to those based on a single computational resource.

6.1 Scheduling Policy Analysis

Under synchronous assumption, synchronous equivalence will hold for some subsets of implementations based solely on the scheduling policy, for any given CFSM specification. A sufficient condition for two implementations to be synchronously equivalent is that they both are implemented on a single computational resource (processor or circuits) and have the same scheduling policy. We need the following definitions to prove this property:

Definition 6.1 (Implementation) *A* CFSM *implementation is a completely determin-*

istic behavior defined by:

- *a network of* CFSM*s, as defined in Chapter 3,*
- *an architectural mapping, consists of*
  - *a scheduling policy, to be defined later,*
  - *possible ranges of execution delays for* CFSM *components,*
- *an optional synthesis directives, defining execution delays for* CFSM *components, if it is not already deterministic.*

Definition 6.2 (Global State Pattern) *Given a* CFSM *implementation, the* **global state pattern** *is the state of all main automata, all input buffer automata and all output buffer automata of the* CFSM *network.*

Definition 6.3 (Stabilization) *An implementation is stable if and only if no change in the global state pattern or output is possible without the application of a new primary input pattern.*

A system that satisfies the synchronous assumption stabilizes at or before the end of a cycle.

A single primary input pattern can stimulate the design and "generate" a sequence of global state patterns until stabilization is reached. A sequence of primary input patterns (i.e. a primary input trace) "generates" a sequence of sequences of global state patterns. A primary input trace also generates a sequence of sequences of *scheduling points.*

Definition 6.4 (Scheduling Point) *A scheduling point is a point in time where some* CFSM *component completes computation, or produces some output.*

A scheduling point corresponds to the point in time when some "scheduling decision" needs to be made. There are often many scheduling points within a single computation phase.

Scheduling points are related to the behavior of the components of the design. When the implementation is stabilized, the computation phase ends. The end of the computation phase corresponds to a scheduling point. The end of interaction phase is also

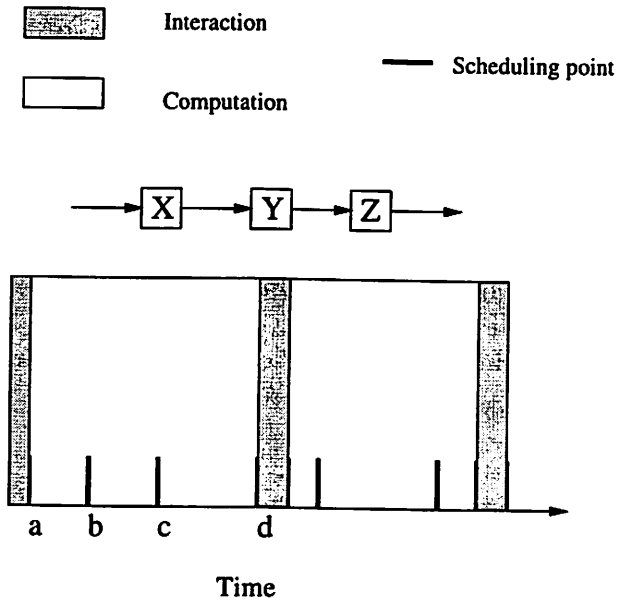

Figure 6.1: Example Illustrating Scheduling Points.

considered a scheduling point because that is when the environment finishes interaction. This concept is demonstrated in Figure 6.1 where scheduling points generated by a simple three buffer chain is shown. Time a is a scheduling point because the environment finishes interaction. At scheduling point b, component X finishes execution and component Y starts to execute because it is the only component with input and consequently the only component that can be scheduled. At time c, component Y finishes execution and component Z starts, so it is another scheduling point. At scheduling point d, Z finishes execution and the computation phase ends because the design has stabilized.

A network of CFSMs, an architectural mapping, and synthesis directives together correspond to an implementation that is fully deterministic. An input trace $i = \{i_1, i_2, ...\}$ can be said to generate a sequence of sequences of scheduling points $\{\{a_1^1, a_1^2, ...\}, \{a_2^1, a_2^2, ...\}, ...\}$ for an implementation where $\{a_1^1, a_1^2, ...\}$ is due to i_1 and $\{a_2^1, a_2^2, ...\}$ is due to i_2. Scheduling points have the following important property:

Lemma 6.1 *Given two implementations, A and B, of the same specification, and an arbitrary input trace* $i = \{i_1, i_2, ...\}$, *i generates a sequence of sequences of scheduling points* $\{\{a_1^1, a_1^2, ...\}, \{a_2^1, a_2^2, ...\}, ...\}$ *for implementation A, and* $\{\{b_1^1, b_1^2, ...\}, \{b_2^1, b_2^2, ...\}, ...\}$ *for implementation B. Let the global state pattern be* $\{\{P_1^1, P_1^2, ...\}, \{P_2^1, P_2^2, ...\}, ...\}$ *for imple-*

mentation A, and $\{\{Q_1^1, Q_1^2, ...\}, \{Q_2^1, Q_2^2, ...\}, ...\}$ *for implementation B at the scheduling points. A and B are synchronously equivalent, if* $P_n^m = Q_n^m$*, for all integer m,n.*

Proof: By definition, the implementation stabilizes at some scheduling point. Since $P_n^m = Q_n^m$ for all scheduling points, they must also be the same at all stabilizing points. In addition, an output can only occur at a scheduling point. Since $P_n^m = Q_n^m$, output patterns are the same at the stabilizing points also. Therefore, outputs are the same at the end of the cycle (stabilization). Therefore, A and B are synchronously equivalent to each other. ■

Definition 6.5 (Scheduling Policy) *A scheduling policy for a* CFSM *network is composed of three functions:* **Enable**, *which defines the condition under which a* CFSM *should be considered for execution.* **Select**, *which defines a subset of the enabled* CFSM*s to actually be executed, and* **Execute**, *which defines the condition under which selected* CFSM*s can start executions.*

A component is usually considered *enabled* for execution if one or more of its inputs have arrived. However, a scheduling policy may enable a component even when no input events are present. In this case, the execution should consume no input, produce no output and leave the component in the same state. This is known as *empty execution.* This feature, and indeed the whole reason for splitting the **Enable** and **Select**, make it much easier to define and analyze worst case communication scenarios. A worst case communication signature can be related to real execution traces by padding execution traces with empty executions. We will use this notion of worst case communication extensively in Chapter 7 and Chapter 8.

An enabled component can be always *selected*, or it can be selected according to some list or priority condition. For a non-preemptive policy, selected component can only *execute* when all previously executing components have completed their executions. Under a preemptive policy, an executing component may be suspended to have a newly selected component executing in its place.

Some of the more common scheduling policies include Unit Delay Parallel (UDP), Static Priority Serial (SPS), and Cyclic Executive Serial (CES).

Definition 6.6 (Unit Delay Parallel) *Unit delay parallel scheduling policy for a* CFSM *network is defined by:*

- *all components are* **Selected**,
- *all selected component are* **Executed** *upon stabilization of all previous executions.*

Definition 6.7 (Static Priority Serial) *Static priority serial scheduling policy for a* CFSM *network is defined by:*

- *A component is* **Enabled** *if one or more of its inputs have arrived,*
- *An enabled component is* **Selected** *by some statically determined priority order.*
- *The selected component is* **Executed** *upon stabilization of the previous execution.*

Definition 6.8 (Cyclic Executive Serial) *Cyclic executive serial scheduling policy for a* CFSM *network is defined by:*

- *A component is* **Enabled** *if one or more of its inputs have arrived,*
- *An enabled component is* **Selected** *by some statically determined list.*
- *The selected component is* **Executed** *upon stabilization of the previous execution.*

UDP, SPS, and CES scheduling policies are *delay insensitive.*

Definition 6.9 (Delay Insensitive Scheduling Policy) *A scheduling policy for a given* CFSM *network is* **delay insensitive** *if different delay assignments to the (*CFSM*) component executions will result in implementations that are guaranteed to be synchronously equivalent to each other.*

For a CFSM network, the UDP scheduling policy essentially implements Moore synchrony. It should not be surprising that all UDP implementations (i.e. implementations with UDP scheduling policy, but possibly with different clock speeds) are synchronously equivalent to each other.

Theorem 6.2 *UDP implementations of the same* CFSM *specification are synchronously equivalent.*

Proof: Given two such implementations A and B and an arbitrary input trace, $P_0^0 = Q_0^0$ because they are specified by the initial state, initial output and the environment. We can now proceed by induction.

- Base Case $P_0^0 = Q_0^0$
- Induction Hypothesis $P_0^i = Q_0^i$
- Prove: $P_0^{i+1} = Q_0^{i+1}$
  $P_0^i = Q_0^i$. UDP dictates that all components are executed in parallel with the same delay. Outputs and next state of each component are computed at the next scheduling point $i + 1$. Since the output and transition relations are identical for A and B, $P_0^{i+1} = Q_0^{i+1}$

At stabilization point j, $P_j = Q_j$. At the next scheduling point, it must be that $P_{j+1}^0 = Q_{j+1}^0$ because they are the same pattern as at previous scheduling point plus primary input, which has to be the same given the same environment.

Therefore, $P_n^m = Q_n^m$ for any integer m and n. Due to Lemma 6.1, the theorem is proven. ∎

The very popular SPS and CES scheduling polices are also delay insensitive.

Theorem 6.3 *Single processor SPS implementations of the same* CFSM *specification and priority order are synchronously equivalent.*

Proof: Given two such implementations A and B and an arbitrary input trace, $P_0^0 = Q_0^0$ because they are specified by the initial state, initial output and the environment. We can now proceed by induction.

- Base Case $P_0^0 = Q_0^0$
- Induction Hypothesis $P_0^i = Q_0^i$
- Prove: $P_0^{i+1} = Q_0^{i+1}$
  Because $P_0^i = Q_0^i$, the SPS implementations with the given priority order make the same scheduling decision and execute corresponding components. Output and next state of that components are calculated for the next scheduling point $i + 1$. Since the output and transition relations are identical for A and B, $P_0^{i+1} = Q_0^{i+1}$.

At each stabilization point j, $P_j = Q_j$. At the next scheduling point, it must be that $P^0_{j+1} = Q^0_{j+1}$ because they are the same pattern as at previous scheduling point plus primary inputs, which have to be the same given the same environment.

Therefore, $P^m_n = Q^m_n$ for any integer m and n. Due to Lemma 6.1, the theorem is proven. ∎

Theorem 6.4 *Single processor CES implementations of the same* CFSM *specification and list order are synchronously equivalent.*

Proof: Given two such implementations A and B and an arbitrary input trace, $P^0_0 = Q^0_0$ because they are specified by the initial state, initial output and the environment. We can now proceed by induction.

- Base Case $P^0_0 = Q^0_0$
- Induction Hypothesis $P^i_0 = Q^i_0$
- Prove: $P^{i+1}_0 = Q^{i+1}_0$

  $P^i_0 = Q^i_0$. the CES implementations with the same list order make the same scheduling decision and execute corresponding components. Output and next state of those components are calculated for the next scheduling point $i+1$. Since the output and transition relations are identical for A and B, $P^{i+1}_0 = Q^{i+1}_0$

At stabilization point j, $P_j = Q_j$. At the next scheduling point, it must be that $P^0_{j+1} = Q^0_{j+1}$ because they are the same pattern as at previous scheduling point plus primary inputs, which have to be the same given the same environment.

Therefore, $P^m_n = Q^m_n$ for any integer m and n. Due to Lemma 6.1, the theorem is proven. ∎

Theorem 6.3 and Theorem 6.4 can be easily extended to preemptive scheduling because for SPS and CES, preemptions always occur at scheduling points. The environment is not allowed to interact with the design during a computation phase and the only event that can occur to "preempt" an execution must be produced by some component in the design. As previously mentioned, production of an output event corresponds to a scheduling point.

In essence, a scheduling policy is delay insensitive if the execution ordering of the components is not dependent on time. An example of scheduling policies that are not delay

insensitive is the class that is referred to as "time-slicing". In "time-slicing", each enabled component is allotted a certain amount of time and if the execution has not been completed within the allotted time, the component will be preempted. For a CFSM network, changing some execution delays of CFSM may result in different time-slicing implementations that are not synchronously equivalent to each other. Multi-processor architectures for a CFSM network are also generally not delay insensitive because the communication between the processors may depend on the speeds of the processors.

If a delay insensitive scheduling policy is chosen at the architectural mapping stage, the previous theorems state that different implementations with that same delay insensitive scheduling policies are synchronously equivalent to each other. Delay insensitive scheduling policies provide the advantage that the designer is free to optimize individual components and the resulting implementation is guaranteed to be synchronously equivalent to the original one. A design with a delay insensitive scheduling policy is functionally robust with respect to uncertainty in timing delays, such as those that may be caused by cache misses. The designer can thus have a high confidence that the dynamically varying delay can affect only the performance of a design, not the functionality of a design. Furthermore, functional simulation of an implementation with a delay insensitive scheduling policy can be done without costly and often inaccurate timing estimation. Delay insensitive scheduling policies take full advantage of the synchronous assumption where function and timing are strictly separated. This property of SPS and CES helps to explain why they are so popular among embedded system designers.

6.1.1 Validating Experiment: Seat Belt Alarm Controller

Performing a scheduling policy analysis is extremely straightforward. If two implementations have the same scheduling policy which is delay insensitive (e.g. UDP, SPS, CES), then they are synchronously equivalent. Otherwise, the analysis result is inconclusive. This analysis can be applied to any design based on CFSM networks. To show that this result is not trivial, we use formal verification tools to perform exact equivalence analysis for a particular design: the seat belt alarm controller from Section 2.1. Implementations with the following scheduling policies are considered:

- UDP

  Fully synchronous hardware (unit delay parallel) implementations.

- SPS-TC

  Single processor implementations with static priority serial scheduling policy. Timer is at higher priority than controller.

- SPS-CT

  Single processor implementations with static priority serial scheduling policy. Controller is at higher priority than timer.

- CES-TC

  Single processor implementations with Round Robin (single appearance cyclic executive serial) scheduling policy. The list is timer, then controller.

- CES-CT

  Single processor implementations with Round Robin scheduling. The list is controller, then timer.

All execution policies are non-preemptive. All delays are set to non-zero, arbitrary, finite, and non-deterministic. By comparing two "scheduling policies" with the formal verification tools, we are essentially comparing all possible delay assignments of one scheduling policy with all possible delay assignments of the other (maybe the same) scheduling policy. All examples are written in BLIF-MV, and the formal verification tool, VIS [BSVA+96], is used to test the synchronous equivalence of the implementations. The experiments are run on a 625MHz Alpha processor with 2 gigabytes of memory. The results are shown in Table 6.1.1. All the results are consistent with our prediction. Notice that although the system is extremely simple, consisting of a 3-state CFSM and a 10-state CFSM, it still takes 21 seconds in the worst case on a very powerful machine.[1] For any realistic design, the result is likely to be inconclusive due to extremely long computation times by the formal verification tools. For this example, the formal verifications all finish within reasonable time and memory requirement. Since the representation is exact, the results are either yes or no (as oppose to yes/inconclusive for abstract analysis). The static scheduling policy analysis is performed by a simple comparison of the scheduling policies and takes essentially no time at all. It achieves the same result for comparing implementations of the same scheduling policy with different delay characteristics (UDP to UDP, SPS-TC to SPS-TC, SPS-CT to SPS-CT, CES-TC to CES-TC, and CES-CT to CES-CT). These are the

[1] Communication buffers, that are non-deterministic, significantly increase the state space size even in this case.

Implementation 1	Implementation 2	Synchronous Equivalent?	CPU time(seconds)
UDP	UDP	Yes	21.0
UDP	SPS-TC	No	12.1
UDP	SPS-CT	No	7.5
UDP	CES-TC	No	14.9
UDP	CES-CT	No	9.5
SPS-TC	SPS-TC	Yes	16.5
SPS-TC	SPS-CT	No	9.9
SPS-TC	CES-TC	Yes	19.2
SPS-TC	CES-CT	No	12.5
SPS-CT	SPS-CT	Yes	13.8
SPS-CT	CES-TC	No	11.8
SPS-CT	CES-CT	Yes	18.1
CES-TC	CES-TC	Yes	20.4
CES-TC	CES-CT	No	12.7
CES-CT	CES-CT	Yes	19.4

Table 6.1: Full Reachability Analysis of Seat Belt Example.

true positives. In the case of comparing SPS-TC with CES-TC, and SPS-CT with CES-CT, the result of static analysis is inconclusive due to false negatives (i.e. exact analysis returns positive). All other combinations of comparison are inconclusive due to true negatives (i.e. exact analysis returns negative). For this example, the implementation space is split into 3 different equivalence classes: 1) UDP, 2) SPS-TC, CES-TC, and 3) SPS-CT, CES-CT. In the next chapter, we will introduce communication analysis, that requires a little more computation time but is able to give us the equivalence between SPS-TC and CES-TC, and between SPS-CT and CES-TC.

6.2 System Graph Analysis

Definition 6.10 (System Graph) *In the system graph of a* CFSM *network, each* CFSM *component is represented by a node and each signal from a sender* CFSM *to a receiver* CFSM *is represented by a directed edge. All primary inputs are represented by a single node with only outgoing edges and all primary outputs are represented by a single node with only incoming edges.*

The system graph for the seat belt example is shown in Figure 2.3. Note that a system graph may be cyclic. In the figure, the single source and sink are split into individual

PI's and PO's for the sake of clarity. A given node in a graph may have many predecessors and many successors.

Definition 6.11 (Predecessor) *A node is the predecessor of another node if it is in its transitive fan-in. The set of nodes that are predecessors of node i is denoted by $Pred(i)$.*

In the seat belt example, $Pred$(controller) and $Pred$(timer) include all nodes except the node that represents primary outputs. A cycle in the system graph makes every node in the cycle both a predecessor and a successor of every other node in the cycle. For the example in Figure 6.1, $Pred(X)$ includes only the primary input. $Pred(Y)$ includes X and the primary input. $Pred(Z)$ includes X, Y, and the primary input.

Definition 6.12 (Successor) *A node is the successor of another node if it is in its transitive fan-out.*

Definition 6.13 (Liveness) *An implementation is* **live** *if and only if any event at the input of components must eventually be consumed by an execution.*

Most interesting implementations of interesting specifications are live. System graphs have the following property:

Theorem 6.5 *Live implementations of the same* CFSM *specification are synchronously equivalent to each other if the system graph without the primary output node is a tree.*

Proof: Since the system graph without the primary output node is a tree, each component has only one input and hence can execute at most once for any set of primary inputs. In fact, it can only execute after its immediate predecessor has executed, regardless of the choice of architectural mapping and synthesis directives. In addition, it must execute because the implementation is live. ∎

The system graph of the seat belt example is not a tree, while the one for the example in Figure 6.1 is. For the design in Figure 6.1, all live implementations satisfying the synchronous assumption are synchronously equivalent to each other.

The reason why some graph may not be a tree is due to *reconvergence*. In a directed tree, there is at most one way to traverse from one node to another. A *reconvergence* exists if there is more than one way to traverse from node a to some node b. Node a will be called a *split-off* point if it has more than one immediate successor and some node b can

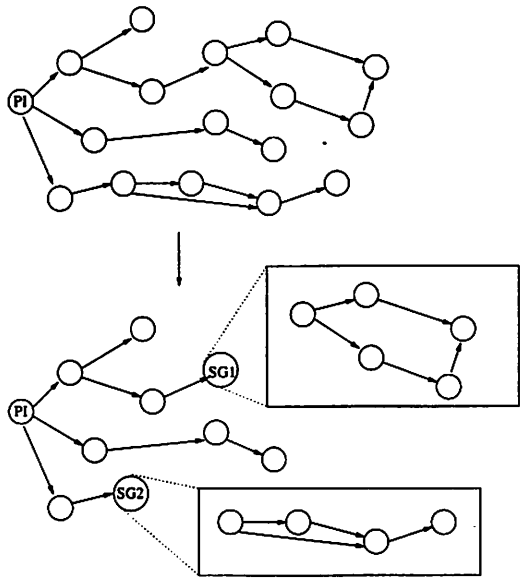

Figure 6.2: Converting into Tree of Nodes and Successor Free Subgraphs.

be reached through more than one successor of a. Our goal is to replace the nodes and edges starting from the *split-off* point by a single node representing a subgraph. Any non-tree system graph can be transformed into a tree of nodes and subgraphs. The following algorithm performs one such transformation. There is no canonical transformation. This transformation produces a graph where the nodes representing subgraphs are successor-free. The starting point is a system graph minus the primary output node:

1. Perform Depth First Search (DFS) from the primary input node. Mark each node as it is visited.

2. If a node previously visited is encountered again, there is a reconvergence. Perform a Breadth First Search (BFS) on the reverse graph to find and mark the split-off point of the reconvergence. Continue with the original DFS.

3. Split-off points represent the root of subgraphs. Discard split-off points that represent subgraphs that are contained in another subgraph represented by another split-off point. This can be done by a second pass of DFS.

An example of the transformation is shown in Figure 6.2. Transforming the system graph of the seat belt example returns the original system graph as the only subgraph in the system.

The next theorem can be used to establish synchronous equivalence hierarchically. It starts from a transformed system graph that is a tree of nodes and successor-free subgraphs. By successor-free we mean that the nodes representing the subgraphs at the top level do not have successors.

Theorem 6.6 *Two live implementations are synchronously equivalent to each other if the system graph without the primary output node can be transformed into a tree of nodes and successor-free subgraphs such that each corresponding subgraph pair is synchronously equivalent.*

Proof: Since the transformed system graph is a tree, all nodes can be executed once and all subgraphs can be "invoked" once. Hence, all corresponding subgraphs of the implementations can be "invoked" once with the same input. Since corresponding subgraphs are synchronously equivalent to each other, they will produce the same outputs and state changes. Other components that are not part of the subgraphs are part of the tree. According to Theorem 6.5, they must produce the same outputs and state changes as well. Therefore, the two implementations are synchronously equivalent to each other. ∎

Theorem 6.6 suggests a hierarchical synchronous equivalence checking algorithm. After splitting the system into a tree of nodes and subgraphs, other algorithms can be used to check the synchronous equivalence of the corresponding subgraphs. Together they imply the synchronous equivalence of the two implementations.

6.3 Mixed Analysis

Obviously, one can utilize both the system graph and the scheduling policies to help in analyzing synchronous equivalence statically.

Theorem 6.7 *A processor implementation with single appearance Cyclic Executive Scheduling (also known as Round Robin) is synchronously equivalent to Unit Delay Parallel implementation if*

- *the system graph is acyclic, and*
- *if $i \in Pred(j)$, then j is earlier in the list than i.*

Proof: Assume a synchronous hardware implementation A, and a single processor implementation with round robin scheduling B of the same specification. Consider an arbitrary input trace and its generated set of scheduling points a and b, respectively, and set of global state/signal pattern P and Q, respectively. At start up, $P_0^0 = Q_0^0$ because they are specified by the initial state, initial output and the environment. Let Q_0^i be the scheduling point corresponding to a single "list" execution of the round robin list. Since all successors are executed before their predecessor, from Q_0^0 to Q_0^i, the input to each executed component at its execution time remains unchanged from startup. Since each component has the opportunity to execute once with the signal pattern of Q_0^0, the state and output pattern will be the same as a single parallel execution, therefore $P_0^1 = Q_0^i$.

Similarly, there exists a j corresponding to the second "list" execution of the round robin list, and $P_0^2 = Q_0^j$. More formally by induction:

- Base Case
  $P_0^0 = Q_0^0$
- Induction Hypothesis
  $P_0^i = Q_0^j$ where j corresponds to the end of the "list".
- Prove: $P_0^{i+1} = Q_0^k$, where k corresponds to end of the next "list" execution
  $P_0^i = Q_0^j$. A single execution of the "list" executes all successors before their predecessor. Not only does that execute each component once, but each component sees Q_0^j at the time of its execution, therefore the executions are not dependent, therefore at the end of the list execution, $P_0^{i+1} = Q_0^k$.

At the stabilization point l, $P_l = Q_l$. At the next scheduling point, P_{l+1}^0 and Q_{l+1}^0 both have the pattern at the previous scheduling point plus the primary inputs, which have to be identical for both implementations also. Therefore $P_{l+1}^0 = Q_{l+1}^0$, and induction applies again.

By induction, since all components execute exactly once for both implementations and with the same input, the primary outputs also agree on this basis, therefore the two implementations are synchronously equivalent. ∎

This theorem suggests a synthesis approach. Given a UDP implementation of an acyclic specification, one can easily come up with an equivalent CES implementation by levelizing the components.

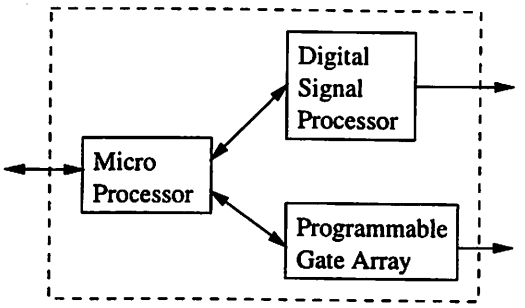

Figure 6.3: Example of Co-Processor Architecture.

6.4 Analysis of Heterogeneous Architectures

One of the salient characteristics of embedded systems is that they may be implemented on heterogeneous architectures. Different parts of the system may be better suited for mapping to computational resources with very different performance and cost factors. Here we consider two very common heterogeneous architectures, the co-processor architecture and the synchronous parallel architecture, and show how they can be modeled for efficient synchronous equivalence analysis.

An example of the co-processor architecture is shown in Figure 6.3. The microprocessor acts as a master of the communication to and from the co-processors. Whenever there is some computation to be done on the co-processors, the microprocessor emits the events and associated data to the co-processors, and waits for the operation on the co-processors to be completed. An acknowledge event is sent back to the master, before continuing with its own operation. The operations of the master and the co-processors thus become serialized. A co-processor architecture can be modeled as, and is indeed synchronously equivalent to, a static priority serial scheduling policy on a single processor architecture with all the components mapped to the co-processors having higher priority than the ones that are mapped to the master.

The synchronous parallel architecture is characterized by processors executing in parallel, but communication is allowed only at a time when all the processing on each processor has been completed. An example is shown in Figure 6.4. A synchronous parallel architecture can be hierarchically modeled as having a unit delay parallel scheduling policy on top of the original single processor scheduling policies for individual processors.

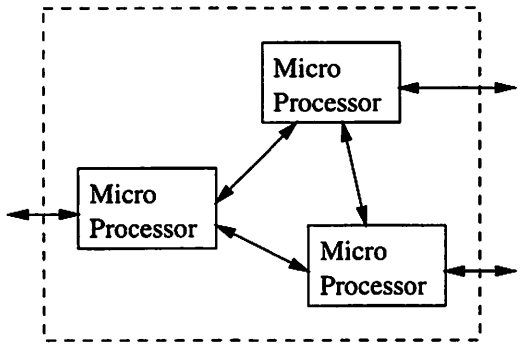

Figure 6.4: Example of Synchronous Parallel Architecture.

6.5 Conclusions

The definition of synchronous equivalence and the application of synchronous assumption lead to very powerful static algorithms where equivalence is conservatively verified by the characteristics of the scheduling policies or the structure of the system graph. However, the result may be inconclusive due to false negatives.

Viewing static equivalence analysis from the standpoint of abstraction/refinement provides useful insight. The analysis at this level is extremely abstract. Lots of information about the system is abstracted away. Utilizing this information will inevitably add to computational complexity of the analysis algorithms. One must refine and include more information in the analysis in a judicious manner to make it worthwhile.

In the next chapter we introduce *Communication Analysis*. The starting point is delay insensitive scheduling policies and the pieces of "information" added are those related to the system graph and abstracted local functions of the components. Communication analysis is no longer considered static because a single pass of execution on the abstract model is used to capture information of the system graph and local functions (i.e. about the communication between the components).

Chapter 7

Communication Analysis

While two implementations with the same delay insensitive scheduling policy are guaranteed to be synchronously equivalent to each other, two implementations with different delay insensitive scheduling policies may still be synchronously equivalent to each other, depending on the behavior of the design. In this chapter, we introduce communication analysis that can be used to conservatively check the equivalence between two different delay insensitive scheduling policies.

There is a simple intuition for looking at the communication between components. Since the corresponding components in the two implementations are guaranteed to have the same functionality, and the connectivity among the components is also guaranteed to be the same, it should thus be possible to deduce the equivalence of two implementations from the behavior of the communication. It should be possible to establish a *communication signature* in the flavor of worst-case analysis in real-time scheduling [LL73]. If two implementations have the same communication signature, they should be synchronously equivalent to each other. By looking at only the worst-case communication characteristics and not at the details, the analysis can be efficient, though at a cost of being conservative. The communication signature is derived from the *execution traces* of an implementation.

7.1 Execution Trace

During a computation phase, there is no interaction between the design and its environment. Within the design, however, components receive events, perform executions in some order, and send out events that can trigger other executions. A given component

can be executed many times during a computation phase.

Definition 7.1 (Local Execution Trace) *Given a legal input trace (i.e. one that satisfies the synchronous assumption), the sequence of input event patterns consumed by a component at each execution is called its* **local execution trace.**

Definition 7.2 (Execution Trace) *Given a legal input trace, the* **execution trace** *of the implementation is the list of all local execution traces of all components.*

An execution trace does not contain any timing information, except in the form of ordering among the sequential atomic executions of the same component. The ordering of the executions of different components is not part of the execution trace. This feature allows widely different implementations to have the same execution traces.

Consider the example in Figure 7.1. For component B, if only m_1 or i_2 is present at the time of execution, o_1 will not be emitted, though the input will be consumed, according to the semantics of the CFSM. Figure 7.2 shows how an execution trace can be obtained through simulation. An SPS implementation with A<B, given primary input $i_1, i_2 = 11$ has B executing first with input $m_1, i_2 = 01$, then A executing with input 1, then B executing again with input 10.

The design in Figure 7.1 is memoryless, so every computation phase has the same response to the same primary input pattern. We present all possible execution traces for three different implementations of the design in Table 7.1. Three implementations are being considered: a single processor SPS with component A at a lower priority than component B, a concurrent hardware UDP implementation where A and B execute in parallel with the same delay, and a single processor SPS with A at a higher priority than B. 1/0 in the table indicates the presence/absence of an event. Recall that the components are reactive, so a non-empty execution will take place only when there is some event "1" at the input.

The execution trace from Figure 7.2 is recorded in the second column, third row of Table 7.1. Time proceeds downward within a cell for the local execution trace of a component, and there is no relationship between local execution traces of different components. For the case of Figure 7.2, there is no ordering information implied between the local execution trace of A: 1, and the local execution trace of B: 01,10. There is, however, an implication that 01 occurs before 10. Information for all the other cells in the table are obtained in a similar fashion as in Figure 7.2.

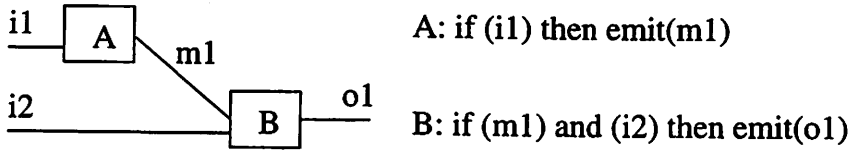

Figure 7.1: Example for Execution Trace.

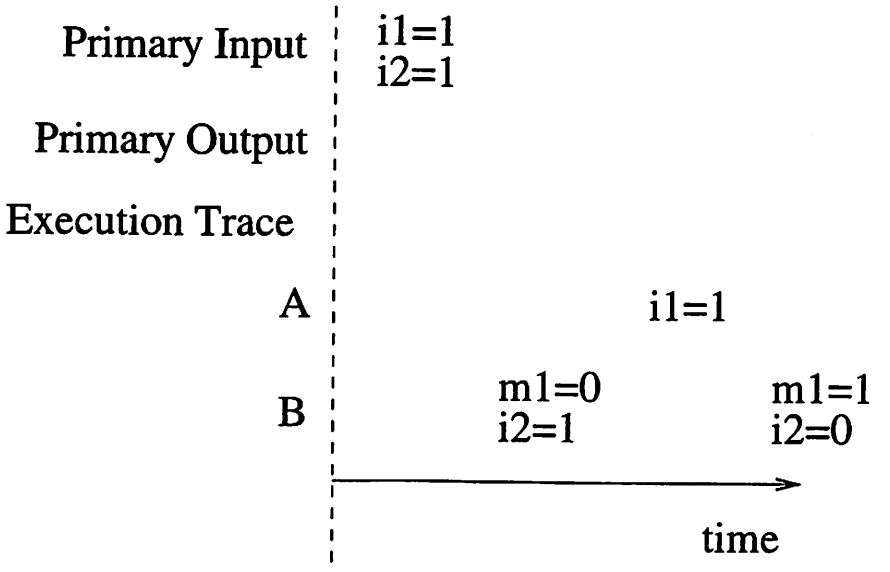

Figure 7.2: An Execution Trace of the Example of Figure 7.1.

An execution trace of the seat belt alarm controller example for a static priority serial scheduling policy with the controller at higher priority than the timer is shown in Figure 7.3. Consider the first cycle. The priority ordering allows the controller to execute first, and produce internal event `Start` to be received and consumed by the timer.

Execution traces have the following important property:

Lemma 7.1 *For every legal input trace (i.e. the synchronous assumption is valid), if the execution traces from two implementations are identical, then the two implementations are synchronously equivalent to each other.*

Proof: Identical execution traces means that the ordered lists of input patterns are the same for all components. Since the outputs of the components and changes in global state patterns are the result of the executions of the components using the sequences of input patterns, they have to be the same for both implementations also. Since output and

	A<B		UDP		A>B	
$i_1 i_2$	A(i_1)	B(m_1,i_2)	A(i_1)	B(m_1,i_2)	A(i_1)	B(m_1,i_2)
11	1	01 10	1	01 10	1	11
10	1	10	1	10	1	11
01		01		01		11

Table 7.1: Execution Traces.

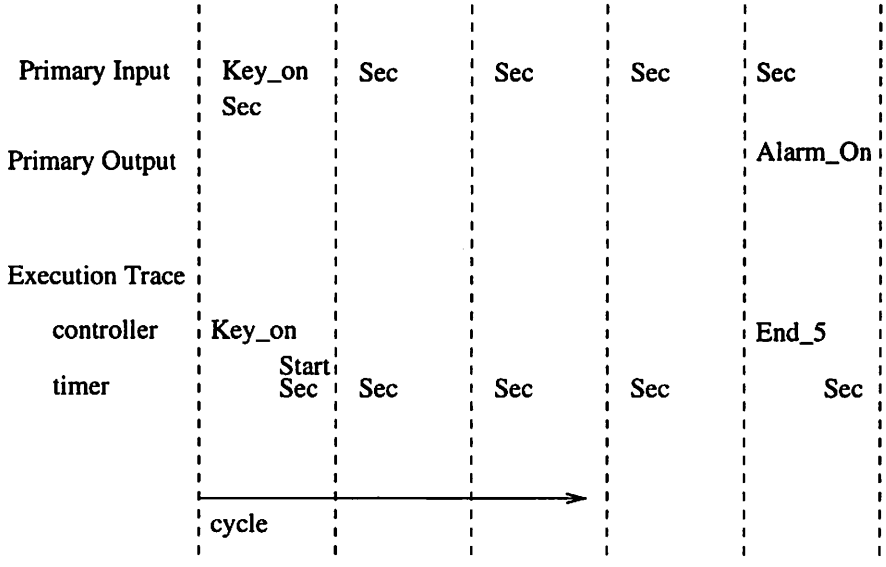

Figure 7.3: An Execution Trace of Seat Belt Example.

global state patterns are the same for all possible primary input traces, they have to be the same at all stabilization points. Since all local execution traces (i.e. input traces) are identical, outputs must be identical at all scheduling points (including stabilizing points). Two implementations are therefore synchronously equivalent to each other. ∎

This lemma suggests a straightforward algorithm for checking synchronous equivalence: simulate all possible input traces and compare the resulting execution traces. Accordingly, implementations A<B and UDP for design in Figure 7.1 are synchronously equivalent to each other. Exhaustive simulation is clearly not practical for all but the most trivial designs. Hence, we introduce *communication analysis*. It is based on the intuition that since two implementations of the same specification have identical component functionalities and

connectivity, the only thing that requires analysis is the communication characteristics. To this end, we look for a "signature" that summarizes the communication.

7.2 Abstracting Communication

We want to abstract, or summarize, communication between the components for a particular implementation. To find a signature that is "worst-case", we use the concept of *non-decreasing* functions defined in a *container space* that is an abstraction of the space of event presence/absence. To find a signature that is correct, though conservative, we utilize the concept of container space *cover* for a event space function. The word "container" is chosen to denote an entity that is necessary, but not sufficient, to contain an actual event. The rationale for this choice will become obvious shortly.

7.2.1 Containers

Intuitively, a container is an entity that may or may not contain an "event". The presence of a container implies the possibility of an event. The absence of a container means that definitely there is not an event. We define a *container set* and an *event set.*

Definition 7.3 (Container Set) *A* **container set,** $\mathtt{C}$, *is a set with two symbols:* $\mathtt{0}$ *and* $\mathtt{x}$. *I.e.* $\mathtt{C} = \{\mathtt{0}, \mathtt{x}\}$.

Definition 7.4 (Event Set) *An* **event set,** E, *is a set with two symbols:* 0 *and* 1. *I.e.* $E = \{0, 1\}$.

Definition 7.5 (Container-Event Set) *A* **container-event set,** $\mathtt{CE}$, *is a set with four symbols:* $\mathtt{0}$, $\mathtt{x}$, 0, *and* 1. *I.e.* $\mathtt{CE} = \{\mathtt{0}, \mathtt{x}, 0, 1\}$.

Notice that the symbols in the container set and the event set are written with different fonts. If the reader finds it hard to distinguish a $\mathtt{0}$ in the container set from a 0 in the event set, he can rest assure that the distinction is only necessary in the strictest mathematical sense. Both of them mean that an event definitely has not occurred. We can define n-dimensional space for each set as follows:

Definition 7.6 (Container Space) *A n-dimensional container space is defined as* $\mathtt{C}^n = \{0, \mathtt{x}\}^n$.

Definition 7.7 (Event Space) *A n-dimensional event space is defined as* $E^n = \{0, 1\}^n$.

Definition 7.8 (Container-Event Space) *A n-dimensional container-event space is defined as* $\mathtt{CE}^n = \{0, \mathtt{x}, 0, 1\}^n$.

A value of 0 for a variable or function in the container space denotes the absence of a container, which corresponds to absence (0) of an event in the event space. A value of x in the container space denotes the presence of the container, which may correspond to either the presence (1) or the absence (0) of an event in the event space. Since the container space defined here is only binary in nature, we could have easily defined all container space variables and container space functions in the Boolean space. However, in Chapter 8, we will extend the container set to be ternary. It is then easier to define a space that is distinct from the Boolean space, once and for all. All the formalism developed in this chapter will also be extended to the ternary container space.

We can define a binary relation, information ordering, between symbols in the container set to be $0 \succeq \mathtt{x}$. 0 has at least as much information as x (i.e. 0 is at least as definite). Information ordering can also be defined in event set and in the container-event set.

Definition 7.9 (Information Ordering) *Information ordering* ($\succeq$) *is defined as a binary relation between two symbols in the event set, container set, and the container-event set:*

- *Event set:*
  $0 \succeq 0; 1 \succeq 1;$
- *Container set:*
  $0 \succeq 0; 0 \succeq \mathtt{x}; \mathtt{x} \succeq \mathtt{x};$
- *Container-event set:*
  $0 \succeq 0; 1 \succeq 1; 0 \succeq 0; 0 \succeq \mathtt{x}; \mathtt{x} \succeq \mathtt{x}; 0 \succeq 0; 0 \succeq \mathtt{x}; 0 \succeq 0; 1 \succeq \mathtt{x}.$

The information ordering can be extended to minterms in spaces.

Definition 7.10 (Minterm) *A single element, m, of a given space S is called a minterm of S, denoted by $m \in S$.*

A minterm of n-dimensional container space, $\mathtt{m} \in \mathtt{C}^n$, is automatically a minterm of n-dimensional container-event space, $\mathtt{m} \in \mathtt{CE}^n$ because container set is contained in the container-event set. The same applies to a minterm in event space. A minterm in container-event space, on the other hand, may not necessarily be a minterm in the container space or the event space.

The definition of information ordering can be extended to event and container space minterms by coordinate-wise extension. For example: $00 \succeq 00$, but $01 \not\succeq 00$; $\mathtt{x0} \succeq \mathtt{xx}$, but $\mathtt{x0} \not\succeq \mathtt{0x}$; $01 \succeq \mathtt{0x}$, but $11 \not\succeq \mathtt{0x}$.

Definition 7.11 (Non-Decreasing) *Given two container space minterms $\mathbf{m_1}$ and $\mathbf{m_2}$, and a container space function $\mathbf{f} : \{0, \mathtt{x}\}^n \rightarrow \{0, \mathtt{x}\}$. $\mathbf{f}$ is* **Non-Decreasing** *if and only if*

$$\mathbf{m_1} \succeq \mathbf{m_2} \Longrightarrow \mathbf{f}(\mathbf{m_1}) \succeq \mathbf{f}(\mathbf{m_2}) \tag{7.1}$$

For a non-decreasing container space function, changing any input from x to 0 will not change its value from 0 to x. This is sometime called the order-preserving mapping.

Similar to the notation for multi-valued logic synthesis [MBNSV93], $\mathbf{f_1}^{\{\mathtt{x}\}}$ is used to denote that container space function $\mathbf{f_1}$ has the value x. Variables are similarly denoted. Thus a value of x for function $\mathbf{F_1}$ with input x0 is denoted as $\mathbf{f_1}(\mathtt{x0})=\mathtt{x}$, or $\mathbf{f_1}^{\{\mathtt{x}\}}(\mathtt{x0})=\mathit{True}$.

Container space function $\mathbf{f_1}^{\{\mathtt{x}\}}=\mathbf{a}^{\{\mathtt{x}\}}+\mathbf{b}^{\{\mathtt{x}\}}$ (meaning that $\mathbf{f_1}$ is x iff a is x or b is x, and 0 otherwise) is non-decreasing. $\mathbf{f_2}^{\{\mathtt{x}\}}=\mathbf{a}^{\{\mathtt{x}\}}*\mathbf{b}^{\{0\}}$ is not non-decreasing in b, because the valuation of $\mathbf{f_2}$ at $\mathbf{a} = \mathtt{x}$ and $\mathbf{b} = \mathtt{x}$ is computed by $\mathbf{f_2}^{\{\mathtt{x}\}}(\mathtt{xx}) = \mathbf{a}^{\{\mathtt{x}\}}(\mathtt{x}) * \mathbf{b}^{\{0\}}(\mathtt{x}) = \mathit{True} * \mathit{False} = \mathit{False}$, and therefore $\mathbf{f_2}(\mathtt{xx}) = 0$. Similarly, we can compute $\mathbf{f_2}^{\{\mathtt{x}\}}(\mathtt{x0}) = \mathbf{a}^{\{\mathtt{x}\}}(\mathtt{x}) * \mathbf{b}^{\{0\}}(0) = \mathit{True} * \mathit{True} = \mathit{True}$, that is $\mathbf{f_2}(\mathtt{x0}) = \mathtt{x}$. Therefore, changing the input from ab=xx to ab=x0 will change output from 0 to x.

Definition 7.12 (Cover) *A container space function $\mathbf{f}$ is a cover of the event space function $f : \{0,1\}^n \rightarrow \{0,1\}$ if:*

- *there is a one to one correspondence between input variables, and*

- *given any event space minterm m and any container space minterm* $\mathbf{m}$

$$m \succeq \mathbf{m} \Longrightarrow f(m) \succeq \mathbf{f(m)} \tag{7.2}$$

Intuitively, or any input minterm $\mathbf{m}$ in the container space, we generate the corresponding event space minterm m by:

- assigning 0 to a event space variable if its corresponding container space variable has the value 0,
- assigning 0 or 1 to a event space variable if its corresponding container space variable has the value x,

then $\mathbf{f}$ is a cover of f if for every m that f evaluates to 1, the corresponding $\mathbf{m}$ evaluates $\mathbf{f}$ to x. Function $\mathbf{f_1}^{\{x\}}=\mathbf{b}^{\{x\}}$ is a non-decreasing cover of $f=\overline{a}*b*\overline{c}$. So are $\mathbf{f_2}^{\{x\}}=\mathbf{b}^{\{x\}}+\mathbf{c}^{\{x\}}$ and $\mathbf{f_3}^{\{x\}}=\mathbf{b}^{\{x\}}+\mathbf{a}^{\{x\}}$. $\mathbf{f_4}^{\{x\}}=\mathbf{a}^{\{x\}}+\mathbf{c}^{\{x\}}$ is a non-decreasing function but is not a cover of f, while $\mathbf{f_5}^{\{x\}}=\mathbf{b}^{\{x\}}+\mathbf{c}^{\{x\}}*\mathbf{a}^{\{0\}}$ is a cover of f but is not non-decreasing.

7.2.2 Well-Behaved Scheduling Policy

Worst case analysis for any general delay insensitive scheduling policy is difficult. We concentrate on algorithms that can be applied to a "well-behaved" subset of delay insensitive scheduling policies.

Definition 7.13 (Well-Behaved Scheduling Policy) *A scheduling policy is* **well-behaved** *if:*

1. *it is delay insensitive,*
2. **Enable** *is true at least for all* CFSM*s with some events on its inputs,*
3. *if* **Enable** *is replaced with an alternative that enables a finite number of additional* CFSM*s, then the execution trace remains the same, except possibly for the insertion of some empty executions.*

Whether or not a given scheduling policy is well-behaved needs to be established separately through formal proofs manually, or with the help of formal verification tools. The proofs will only need to be done once, since the property is not design dependent. Many common scheduling policies are well-behaved. The next three theorems show that UDP, SPS, and CES, are all examples of well-behaved scheduling policies.

Theorem 7.2 *The Unit Delay Parallel scheduling policy for a* CFSM *network is well-behaved.*

Proof: From Theorem 6.2, UDP is delay insensitive. CFSMs are reactive, so executing components without input events produces empty executions. Since UDP executes all components in parallel, execution of one component does not affect the delay of others. Communication only occurs at the end of atomic execution. Since empty execution does not produce any output, it will not communicate anything to other components. Adding more component to the **Enable** function therefore does not change the execution except for the addition of empty executions. UDP is therefore well-behaved. ■

Theorem 7.3 *The Static Priority Serial scheduling policy for a* CFSM *network is well-behaved.*

Proof: From Theorem 6.3, SPS is delay insensitive. CFSMs are reactive, so executing components without input events produces empty executions. For SPS, the result of the **Select** function depends on what is enabled, and on the priority list. By additionally enabling components, **Select** can choose one of the additionally enabled component and perform an empty execution. Since there is only a finite number of additional enabled components, **Select** must eventually select the originally selected component after a finite number of empty executions. Therefore, the execution trace is the same modulo those additionally enabled components (corresponding to empty executions). SPS is therefore well-behaved. ■

Theorem 7.4 *The Cyclic Executive Serial scheduling policy for a* CFSM *network is well-behaved.*

Proof: From Theorem 6.4, CES is delay insensitive. CFSMs are reactive, so executing components without input events produces empty executions. Since the **Select** function of CES is not dependent on what is enabled, changing the **Enable** function does not change the selection and execution except for adding some empty executions. CES is therefore well-behaved. ■

An example of a policy that is not well-behaved is the following (quite un-natural) *Select* rule: "If CFSM C is enabled, the (dynamic) priority is C>B>A. Otherwise, the priority is A>B." At some point in time when B and A are enabled, additionally enabling

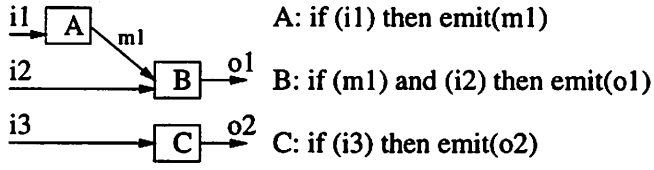

Figure 7.4: Example for Well-Behaved Scheduling Policy.

	WB			WB mod			NWB			NWB mod		
i_1i_2i3	A	B(m1,i2)	C	A	B(m1,i2)	C	A	B(m1,i2)	C	A	B(m1,i2)	C
110	1	01 10		1	01 10	0	1	11		1	01 10	0
100	1	10		1	10		1	11		1	10	
010		01			01			11			01	
111	1	01 10	1	1	01 10	1	1	01 10	1	1	01 10	1
101	1	10	1	1	10	1	1	l0	1	1	10	1
011		01	1		01	1		01	1		01	1
001			1			1			1			1

Table 7.2: Execution Traces for Figure 7.4.

C (which will cause an empty execution on C) can actually change the execution traces of B and A. Figure 7.4 and Table 7.2 illustrate that this scheduling policy is not well-behaved. In the example, we additionally enable C whenever A and B are both enabled. Columns titled "WB" and "WB mod" correspond to an SPS (A<B<C) scheduling policy (a well-behaved scheduling policy). "NWB" and "NWB mod" correspond to the non well-behaved scheduling policy described above. "WB mod" and "NWB mod" correspond to additionally enabling C whenever A and B are both enabled. "WB mod" is identical to "WB" except for the empty execution represented by 0 for the local execution trace of component C. We can see that for the scheduling policy that is not well-behaved, even after taking into account the empty transitions, the execution traces for "NWB" and "NWB mod" are still very different.

7.2.3 Execution Covers

A good communication signature must be easy to compute, so that it can be used in the inner loop of some automatic design exploration procedure. It must also have the property that if two implementations have the same communication signature, they must be synchronously equivalent to each other. For this purpose we introduce a set of *execution covers* (ECs). An execution cover of a given CFSM network is a directed acyclic graph. EC nodes can be thought of as " containers" representing *possible* events, in the flavor of event graphs representing partially order histories [NPW81, McM93], but using the additional notion of possibility to further abstract it. Thus, a container can contain either a "0" (event absence) or a "1" (event presence). Each node is labeled with the corresponding signal. The EC construction procedure also labels each container with a level, but this label is discarded after the construction is completed. Roughly speaking, edges in the EC represent dependencies between input and output events, as well as the ordering among events belonging to the same signal.

The EC for an implementation with a given scheduling policy can be obtained by "simulating", "abstractly executing", or "unfolding" the system graph using the following *Execution Cover Generation Procedure*:

1. Create a container (an x) for each primary input and label it with level 0. Set the current level to 0.

2. Determine the set of *active* inputs for each (CFSM) node. A CFSM input is active if there exists a corresponding container with a level that is larger than that used by previous execution level of that CFSM (i.e. there exists a container that has not been "consumed" by the previous execution).

3. Let all CFSMs with at least one active input be enabled. If no CFSM is enabled, then STOP.

4. Apply the *Select* function of the original scheduling policy to choose the CFSMs to be "executed".

5. *Abstractly execute* the selected CFSMs by increasing the current level by 1, and evaluate a non-decreasing cover of each CFSM output function with all active inputs set to x and other inputs set to 0. If the non-decreasing cover evaluates to x, then

- create a new container and label it with the current level and the name of that output,
- for every active input: create an edge from the most recent container corresponding to that input, to the newly created container,
- create an edge from the previous container labeled with the same name (if any exists) to the new container.

6. "de-activate" the inputs of the executed CFSMs. Go to Step 2.

We can (conservatively) check two implementations for synchronous equivalence by comparing their ECs, as stated by the following result. The theorem applies to any two execution covers, no matter how each of them are generated according to Step 5 above. The proof of this important theorem is postponed until Section 7.2.6.

Theorem 7.5 *If two implementations of a given* CFSM *network with well-behaved scheduling policies have identical ECs, then they are synchronously equivalent.*

The EC generation procedure may not terminate, even if the CFSM network stabilizes for every input pattern. This non-termination is due to looping through the same patterns, and thus can be easily identified in the algorithm implementation. In this case, the algorithm can be aborted and the communication analysis returns inconclusive result. However, if the procedure terminates successfully, then identical ECs implies synchronous equivalence. The abstract nature of EC gives it efficiency, but also makes it reach many inconclusive results due to the false negatives or non-termination. We will show the usefulness of EC on industrial designs.

The abstract execution in Step 5 is performed by evaluating a *non-decreasing cover* of the CFSM output function. There are many non-decreasing covers of a function, so there is a whole set of different ECs that can be used as communication signatures. We present only the Greatest Execution Cover (GEC) and the Least Execution Cover (LEC). GEC is very easy to understand and compute, and also shows that more than one cover is possible. LEC is only slightly more complex and is more useful in practice. It is also least abstract, in a sense that will be explained in Section 7.2.5.

7.2.4 Greatest Execution Cover

The Greatest Execution Cover is computed by creating a new container in Step 5 whenever a CFSM is selected, regardless of what active inputs there may be. Due to the reactive nature of the CFSM, the weakest criterion for a CFSM to be selected is that some event be possibly present at some input. We first convert all signals consisting of an event part and a value part into vectors of pure events. This is done through encoding, usually in a one-hot fashion. An example of this conversion is shown in Section 7.2.4.3. For large systems, this may be quite expensive. Some form of value abstraction similar to those introduced in Chapter 4 may be necessary. The greatest cover function $\mathbf{F}$ in the container space is a disjunction of all the input variables having the value x:

$$\mathbf{F}^{\{x\}} = \sum_{i=1}^{n} \mathbf{x_i}^{\{x\}} \tag{7.3}$$

$\mathbf{F}$, the greatest non-decreasing cover of F, is non-decreasing because changing some input from x to 0 cannot cause the function to change from 0 to x. The output is 0 only if all inputs are 0. The only way to change that output to an x is by changing some input from 0 to x. We can show that $\mathbf{F}$ covers F as follows. There is only one minterm of $\mathbf{F}$ that does not evaluate to an x: the one with inputs all at 0. A minterm in the container space can correspond to many minterms in the event space in general. This minterm of $\mathbf{F}$, however, corresponds to only one minterm of F, namely, the one with inputs all at 0. Due to the reactive nature of CFSM, all 0 at the inputs cannot produce a 1 at the output. $\mathbf{F}$ therefore covers F. It is the greatest, in the sense that it contains all the minterms of all the other covers of the reactive function F. The single all 0 input is not necessary to cover any reactive function.

To abstractly execute a CFSM and obtain the greatest execution cover, we evaluate its greatest non-decreasing cover with all active inputs set to x and all others set to 0.

7.2.4.1 Simple Illustrating Example

The greatest non-decreasing cover for the output function of the CFSMs in the example in Figure 7.1 is:

$$\begin{aligned} \mathbf{m_1}^{\{x\}} &= \mathbf{i_1}^{\{x\}} \\ \mathbf{o_1}^{\{x\}} &= \mathbf{m_1}^{\{x\}} + \mathbf{i_2}^{\{x\}} \end{aligned} \tag{7.4}$$

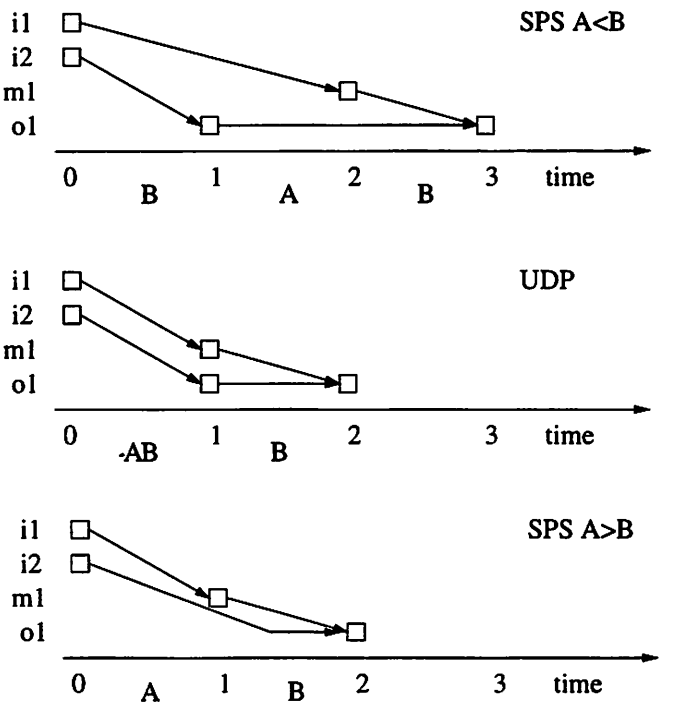

Figure 7.5: Greatest Execution Cover for Example in Figure 7.1.

The Greatest Execution Covers for the example in Figure 7.1 are shown in Figure 7.5 for the given three different execution policies, SPS A>B, UDP, and SPS A<B. The GECs are identical for scheduling policies A<B and UDP, and the two scheduling policies indeed do have identical execution traces as was shown in Table 7.1. According to Lemma 7.1, they are synchronously equivalent to each other. If the GECs are different, as they are between UDP and A>B, we cannot conclude whether the scheduling policies are or are not synchronously equivalent, due to the fact that the analysis is abstract and the possibility of a false negative result cannot be automatically excluded.

7.2.4.2 Shock Absorber Example

We applied the GEC analysis to a real-life industrial design: the shock absorber controller from Chapter 4. The controller sets the shock absorbers' motors to appropriate absorption levels according to inputs from a vertical acceleration (vibration) sensor and a speed sensor. The system graph for this design is shown in Figure 7.6. It is a simplified version of the one in Chapter 4 for the sake of simplicity in visual presentation. In this

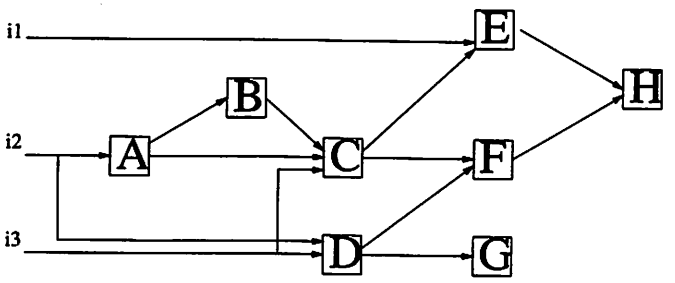

Figure 7.6: System Graph for Shock Absorber Controller.

simplified model, the system receives input i_1, which represents the current shock absorber position, i_2, which represents the wheel motion (present once every wheel revolution), and i_3, which indicates the vertical acceleration. Component A calculates the speed of the automobile. B calculates the acceleration. C calculates a shock absorption strategy according to the speed and acceleration. E calculates the recommended absorption level according to the current position and the one calculated from speed and acceleration. D computes the vertical acceleration and sends it to G. F determines whether or not there is an error condition, from speed, acceleration, or vertical acceleration. H utilizes all the information and computes the next absorption strategy.

We use communication analysis algorithm to decide synchronous equivalence among the following five scheduling policies:

1. Synchronous Hardware (UDP).
2. Single processor with list scheduling (CES): A,B,C,D,E,F,G,H.
3. Single processor with list scheduling (CES): H,G,F,E,D,C,B,A.
4. Single processor with priority (SPS): A>B>C>D>E>F>G>H.
5. Single processor with priority (SPS): A<B<C<D<E<F<G<H.

We obtained the greatest execution covers for all five scheduling policies in Figures 7.7, 7.8, 7.9, 7.10, and 7.11. Lower case "b" represents the output containers form component "B" and "a-B" represents the output containers from component "A" to be received by component "B". For ease of understanding, we also label the "time" axis with both the "level" in the EC generation algorithm and the components executing at that

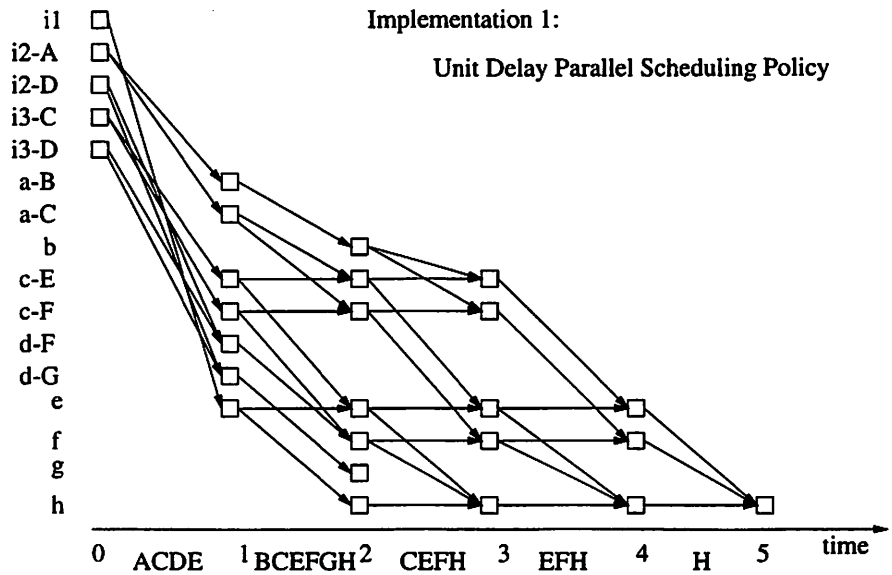

Figure 7.7: Greatest Execution Covers for Shock Absorber Scheduling Policy 1.

level. When checking two ECs for identity, it is not necessary to compare the levels. Only the partial ordering as implied by the edges matters.

From the figures, we can conclude by checking labeled graph isomorphism that scheduling policies 1 and 3 are synchronously equivalent to each other because their GECs are identical. Combining this result with Theorems 6.2, 6.3, and 6.4, we can conclude that any synchronous hardware implementation and any single processor implementation (with any delay characteristics) with the given CES ordering are synchronously equivalent to each other. If they both satisfy timing constraints, a software implementation may have a lower cost and a circuit implementation may have better performance in terms of timing. We can also conclude similarly that scheduling policies 2 and 4 are synchronously equivalent to each other. The conservative analysis was performed in a very short computing time and with negligible memory occupation.

7.2.4.3 Seat Belt Alarm Controller Example

We next attempt to apply GEC communication analysis to the seat belt controller example. We first convert the valued event $Alarm(Boolean)$ into two pure events: $Alarm\_On$ and $Alarm\_Off$. The greatest non-decreasing cover for the output function is:

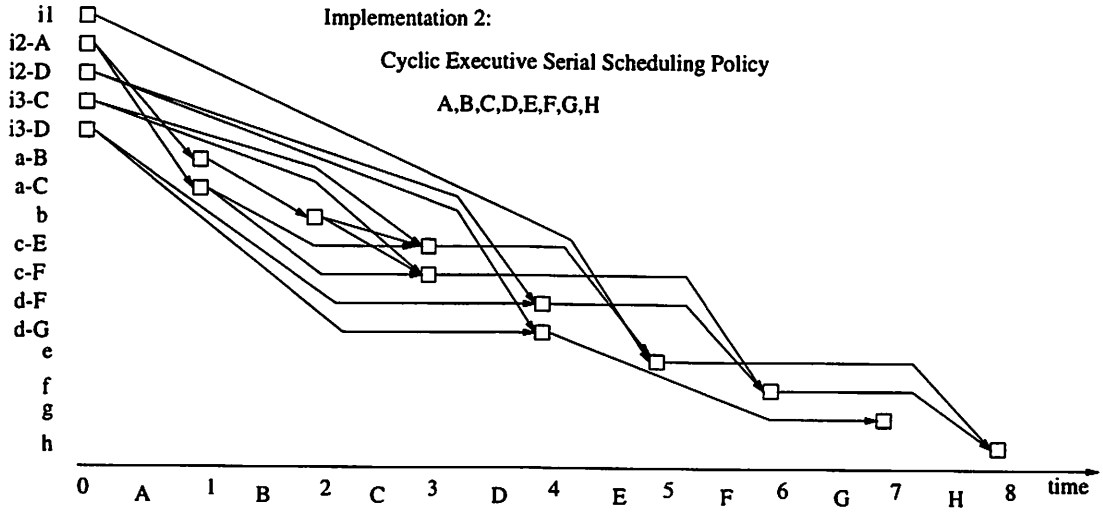

Figure 7.8: Greatest Execution Covers for Shock Absorber Scheduling Policy 2.

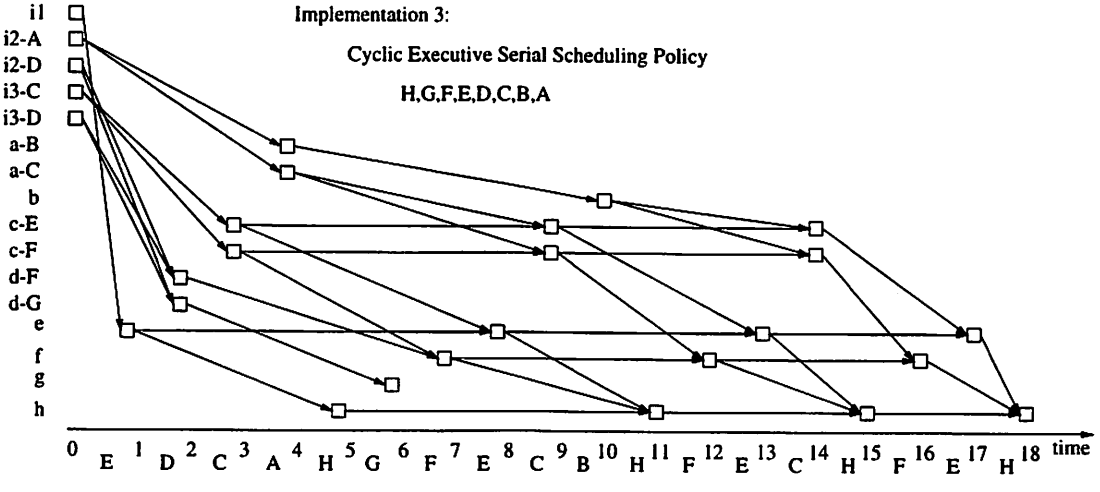

Figure 7.9: Greatest Execution Covers for Shock Absorber Scheduling Policy 3.

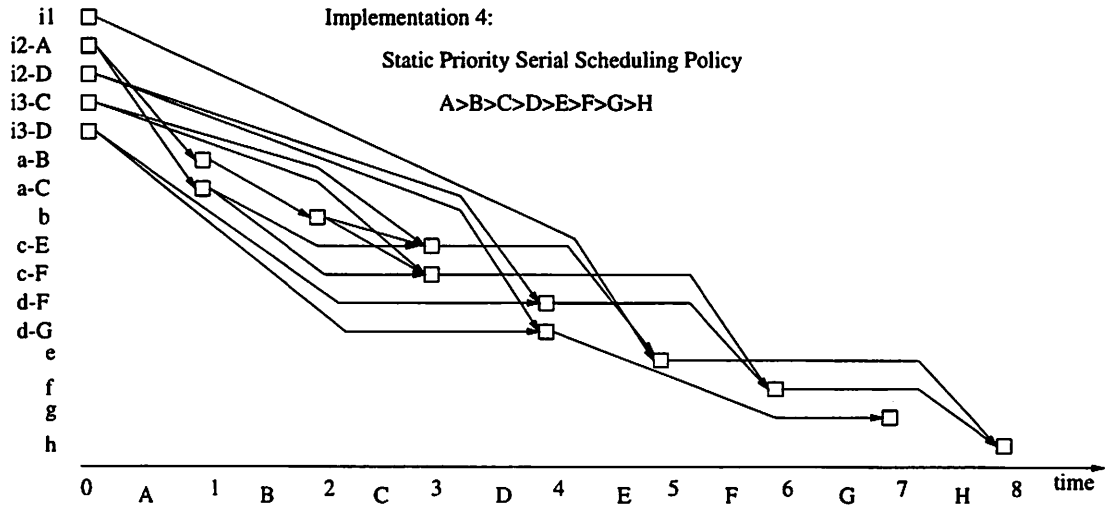

Figure 7.10: Greatest Execution Covers for Shock Absorber Scheduling Policy 4.

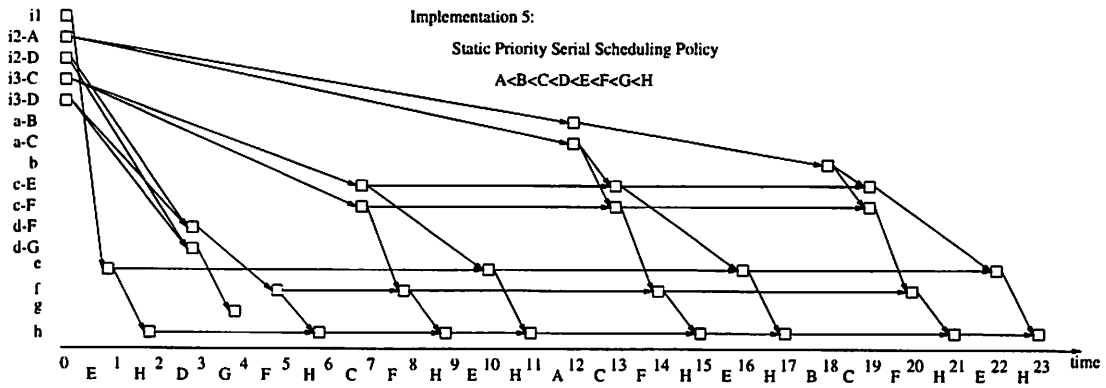

Figure 7.11: Greatest Execution Covers for Shock Absorber Scheduling Policy 5.

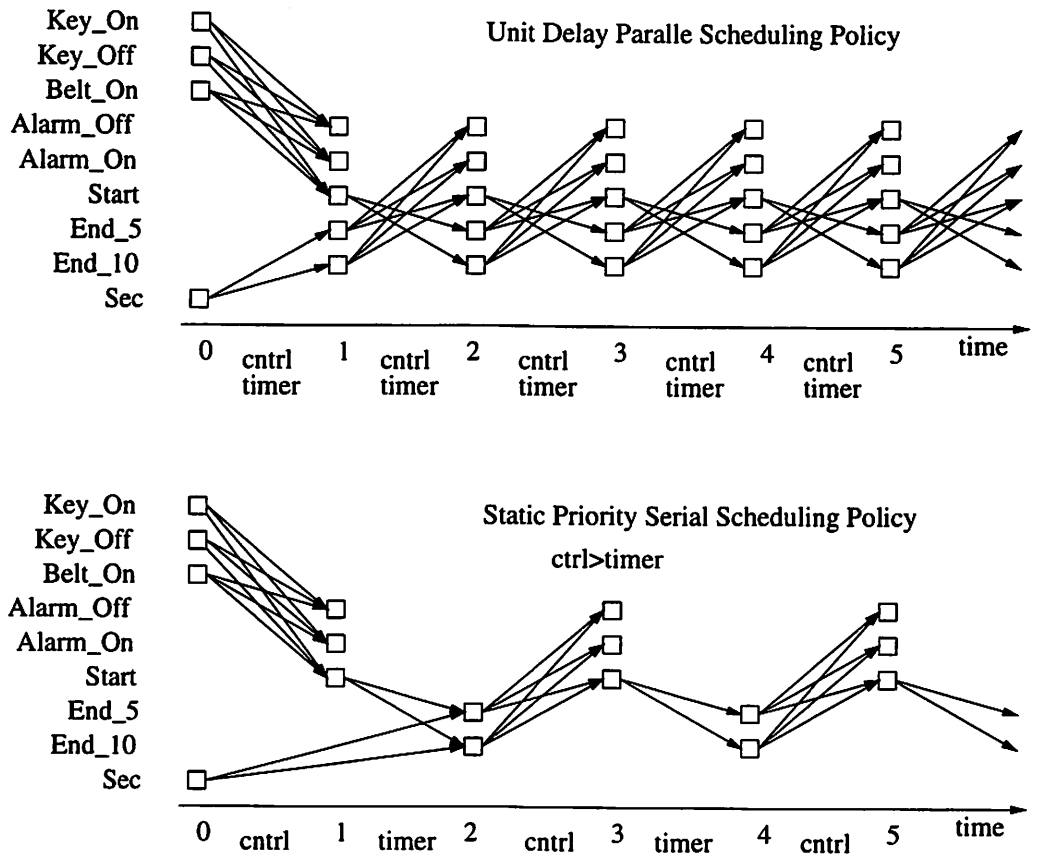

Figure 7.12: Greatest Execution Cover for Seat Belt Controller.

$$
\begin{aligned}
\mathbf{Alarm\_On}^{\{x\}} &= \mathbf{Key\_on}^{\{x\}} + \mathbf{Key\_Off}^{\{x\}} + \mathbf{Belt\_On}^{\{x\}} + \mathbf{End\_5}^{\{x\}} + \mathbf{End\_10}^{\{x\}} \\
\mathbf{Alarm\_Off}^{\{x\}} &= \mathbf{Key\_on}^{\{x\}} + \mathbf{Key\_Off}^{\{x\}} + \mathbf{Belt\_On}^{\{x\}} + \mathbf{End\_5}^{\{x\}} + \mathbf{End\_10}^{\{x\}} \\
\mathbf{Start}^{\{x\}} &= \mathbf{Key\_on}^{\{x\}} + \mathbf{Key\_Off}^{\{x\}} + \mathbf{Belt\_On}^{\{x\}} + \mathbf{End\_5}^{\{x\}} + \mathbf{End\_10}^{\{x\}} \\
\mathbf{End\_5}^{\{x\}} &= \mathbf{Start}^{\{x\}} + \mathbf{Sec}^{\{x\}} \\
\mathbf{End\_10}^{\{x\}} &= \mathbf{Start}^{\{x\}} + \mathbf{Sec}^{\{x\}}
\end{aligned} \tag{7.5}
$$

The GECs for the seat-belt example are shown in Figure 7.12 for two different scheduling policies. Unfortunately, the GEC is infinite for both cases. One abstract execution is as follows: primary inputs first cause the controller to emit a container at **Start**, which causes the timer to execute and emit **End_5** and **End_10**, which in turn cause the controller to execute and emit **Start** again. The GEC is infinite and the analysis result

inconclusive. In the next section, we introduce the Least Execution Cover that abstracts away less information, so that the controller does not emit **Start** when it receives only **End_5** and **End_10** at its input. LEC will in fact generate finite ECs for some systems with loops, while any system with loops will have infinite GECs.

7.2.5 Least Execution Covers

To obtain the Least Execution Cover, the abstract execution in Step 5 of the EC generation algorithm is performed by evaluating the *least non-decreasing cover* of the CFSM output function. To obtain this cover function, we similarly existentially quantify state variables from the output function, to obtain event space function F that depends only on input event space variables $x_1, \ldots x_n$. We then manipulate, in the event space, the function F to obtain an auxiliary function $\mathcal{F}$ also in the event space. $\mathcal{F}$ is then *translated*, one to one, into the container space to obtain cover function $\mathbf{F}$.

Definition 7.14 (Translation) *A container space function* $\mathbf{F}$ *is a translation of a event space function* $\mathcal{F}$ *if and only if, under a given one to one correspondence between input variables, if we*

- *assign* 0 *to a container space variable if its corresponding event space variable has the value* 0,

- *assign* x *to a container space variable if its corresponding event space variable has the value* 1,

then $\mathbf{F}$ *evaluates to* x *if and only if the output of* $\mathcal{F}$ *is* 1.

$\mathbf{F^{\{x\}}}=\mathbf{a^{\{x\}}}+\mathbf{b^{\{x\}}}$ is the translation of the event space function $\mathcal{F} = a + b$.

We want an auxiliary function that has the following property: every minterm in the on-set of F must also be in the on-set of $\mathcal{F}$. In addition, if a minterm with variable $x_i = 0$ is in the on-set of $\mathcal{F}$, then the same minterm, but with variable $x_i = 1$, must also be in the on-set of $\mathcal{F}$. This can be express by the following formulae, for functions with 1, 2,

or 3 input variables:

$$\begin{aligned}
\mathcal{F}(x_1) &= F + F_{\overline{x_1}} \\
\mathcal{F}(x_1, x_2) &= F + F_{\overline{x_1}} + F_{\overline{x_2}} + F_{\overline{x_1 x_2}} \\
\mathcal{F}(x_1, x_2, x_3) &= F + F_{\overline{x_1}} + F_{\overline{x_2}} + F_{\overline{x_3}} + F_{\overline{x_1 x_2}} + F_{\overline{x_1 x_3}} + F_{\overline{x_2 x_3}} + F_{\overline{x_1 x_2 x_3}} \\
&\cdots
\end{aligned} \tag{7.6}$$

The computation of $\mathcal{F}$ can be simplified drastically with the following recursive formulation for a function with n input variables, thanks to the commutativity of cofactors:

$$\begin{aligned}
\mathcal{F}^0 &= F \\
\mathcal{F}^i &= \mathcal{F}^{i-1} + \mathcal{F}^{i-1}_{\overline{x_i}} \qquad \text{for } i = 1, \ldots, n \\
\mathcal{F} &= \mathcal{F}^n
\end{aligned} \tag{7.7}$$

For the case of F having three input variables, the recursive computation is carried out as follows:

$$\begin{aligned}
\mathcal{F} &= \mathcal{F}^3 \\
&= \mathcal{F}^2 + \mathcal{F}^2_{\overline{x_3}} \\
&= \mathcal{F}^1 + \mathcal{F}^1_{\overline{x_2}} + (\mathcal{F}^1 + \mathcal{F}^1_{\overline{x_2}})_{\overline{x_3}} \\
&= \mathcal{F}^1 + \mathcal{F}^1_{\overline{x_2}} + \mathcal{F}^1_{\overline{x_3}} + \mathcal{F}^1_{\overline{x_2 x_3}} \\
&= \mathcal{F}^0 + \mathcal{F}^0_{\overline{x_1}} + (\mathcal{F}^0 + \mathcal{F}^0_{\overline{x_1}})_{\overline{x_2}} + (\mathcal{F}^0 + \mathcal{F}^0_{\overline{x_1}})_{\overline{x_3}} + (\mathcal{F}^0 + \mathcal{F}^0_{\overline{x_1}})_{\overline{x_2 x_3}} \\
&= \mathcal{F}^0 + \mathcal{F}^0_{\overline{x_1}} + \mathcal{F}^0_{\overline{x_2}} + \mathcal{F}^0_{\overline{x_1 x_2}} + \mathcal{F}^0_{\overline{x_3}} + \mathcal{F}^0_{\overline{x_1 x_3}} + \mathcal{F}^0_{\overline{x_2 x_3}} + \mathcal{F}^0_{\overline{x_1 x_2 x_3}} \\
&= \mathcal{F}^0 + \mathcal{F}^0_{\overline{x_1}} + \mathcal{F}^0_{\overline{x_2}} + \mathcal{F}^0_{\overline{x_3}} + \mathcal{F}^0_{\overline{x_1 x_2}} + \mathcal{F}^0_{\overline{x_1 x_3}} + \mathcal{F}^0_{\overline{x_2 x_3}} + \mathcal{F}^0_{\overline{x_1 x_2 x_3}} \\
&= F + F_{\overline{x_1}} + F_{\overline{x_2}} + F_{\overline{x_3}} + F_{\overline{x_1 x_2}} + F_{\overline{x_1 x_3}} + F_{\overline{x_2 x_3}} + F_{\overline{x_1 x_2 x_3}}
\end{aligned} \tag{7.8}$$

The least non-decreasing cover $\mathbf{F}$ is obtained as the translation of $\mathcal{F}$ in the container space:

$$\mathbf{F} = \mathit{Translation}(\mathcal{F}) \tag{7.9}$$

To see that $\mathbf{F}$ is non-decreasing (changing some input from x to 0 cannot cause the function to change from 0 to x), we only have to see that changing some input from 1 to 0 cannot

cause $\mathcal{F}$ to change from 0 to 1. By the definition of translation, **F** is non-decreasing. To see that **F** covers F, we note that for each minterm in the on-set of F with variable x_i=0, the same minterm and with the one with x_i=1 will both be in the on-set of $\mathcal{F}$. This means that the corresponding minterm with $\mathbf{x_i}$=x will evaluate **F** to x, a sufficient condition for **F** to cover F. To see that it is the least such function, we note that taking away any set of minterms of $\mathcal{F}$ which is also in F will make **F** not a cover, because it will make it such that changing some input from 1 to 0 will cause $\mathcal{F}$ to change from 0 to 1. By the definition of translation, this would make **F** decreasing.

To abstractly execute a CFSM, we evaluate its least non-decreasing cover with all active inputs set to x and all others set to 0.

7.2.5.1 Simple Illustrating Example

The output function of the CFSMs in the example in Figure 7.1 is as follows:

$$\begin{aligned} m_1 &= i_1 \\ o_1 &= m_1 * i_2 \end{aligned} \tag{7.10}$$

The least non-decreasing cover for this function is:

$$\begin{aligned} \mathbf{m_1}^{\{x\}} &= \mathbf{i_1}^{\{x\}} \\ \mathbf{o_1}^{\{x\}} &= \mathbf{m_1}^{\{x\}} * \mathbf{i_2}^{\{x\}} \end{aligned} \tag{7.11}$$

The LECs for the example in Figure 7.1 are shown in Figure 7.13 for the given three different scheduling policies, SPS A>B, UDP, and SPS A<B. The LECs are identical for scheduling policies A<B and UDP, and the two scheduling policies indeed produce identical execution traces as was shown in Table 7.1. According to Lemma 7.1, they are synchronously equivalent to each other. If the LECs are different, as they are between UDP and A>B, we cannot conclude whether the implementations are or are not synchronously equivalent, due to the possibility of inconclusive result being caused by false negatives because the analysis is abstract. Notice the containers without outgoing edges in the figure. They are the containers that participate in an execution but the execution did not produce an output container. There are obviously fewer containers in LECs than GECs. The analysis is less abstract at a cost of negligible increase in computation time.

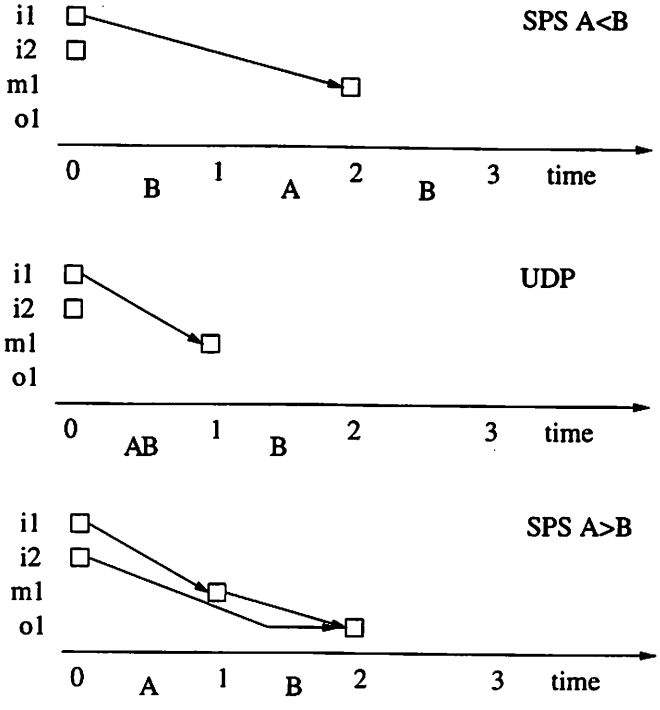

Figure 7.13: Least Execution Cover for Example in Figure 7.1.

7.2.5.2 Seat Belt Alarm Controller Example

We apply LEC analysis to the seat belt controller example. Recall that GEC analysis produced inconclusive result because the execution cover is infinite, and static scheduling policy analysis from Section 6.1 produced conservative result as compared to exact analysis. After converting the valued events to pure events through one-hot encoding and existentially quantifying out state variables, the output function is:

$$\begin{aligned} Alarm\_On &= \overline{Key\_Off} * \overline{Belt\_On} * End\_5 \\ Alarm\_Off &= Key\_Off + Belt\_On + End\_10 \\ Start &= Key\_On * \overline{Key\_Off} * \overline{Belt\_On} \\ End\_5 &= \overline{Start} * Sec \\ End\_10 &= \overline{Start} * Sec \end{aligned} \tag{7.12}$$

Now, the least non-decreasing cover for this function is:

$$
\begin{aligned}
\mathbf{Alarm\_On}^{\{x\}} &= \mathbf{End\_5}^{\{x\}} \\
\mathbf{Alarm\_Off}^{\{x\}} &= \mathbf{Key\_Off}^{\{x\}} + \mathbf{Belt\_On}^{\{x\}} + \mathbf{End\_10}^{\{x\}} \\
\mathbf{Start}^{\{x\}} &= \mathbf{Key\_on}^{\{x\}} \\
\mathbf{End\_5}^{\{x\}} &= \mathbf{Sec}^{\{x\}} \\
\mathbf{End\_10}^{\{x\}} &= \mathbf{Sec}^{\{x\}}
\end{aligned}
\tag{7.13}
$$

The LECs for the seat belt example are shown in Figure 7.14 for several different scheduling policies. For the sake of readability, the LEC for $SPS : T > C$ (Timer at higher priority than controller) has been compressed so that there is an edge from all (5) inputs to all (3) outputs.

The LECs for the seat belt example are finite, unlike its GECs. From the LECs, we can conclude that implementations with $SPS : C > T$ are synchronously equivalent to implementations with $CES : C,T$. We could not make that conclusion from static analysis alone. In fact, for the five sets of implementations considered in Section 6.1.1, LEC analysis returns the same result as exhaustive simulation in the context of the design exploration methodology described in Section 5.3. It does so with negligible time and memory requirement.

7.2.5.3 ATM Switch Example

We applied LEC communication analysis to another real-life industrial design. The algorithm block of a server that supports ATM-based Virtual Private Networks. The complete server [CDV94, FLL+98] required a design effort of approximately 3 man-years. The algorithm block was re-specified as 1200+ lines of Esterel code separated into 13 different CFSMs. If we were to represent even just the control portion of the system (excluding tables) as a Boolean network, it would have required more than 500 binary latches. Without extensive manual abstraction, verifying this design is clearly beyond the capability of existing formal verification tools [BSVA+96].

The algorithm block decides which input cells must be accepted or discarded to avoid node congestion, and implements the shaping and bandwidth partition functions among ATM VPCs. Figure 7.15 provides a functional description of the algorithm block. Upon arrival of a new cell, it receives the Cell ID from the address-lookup module. If

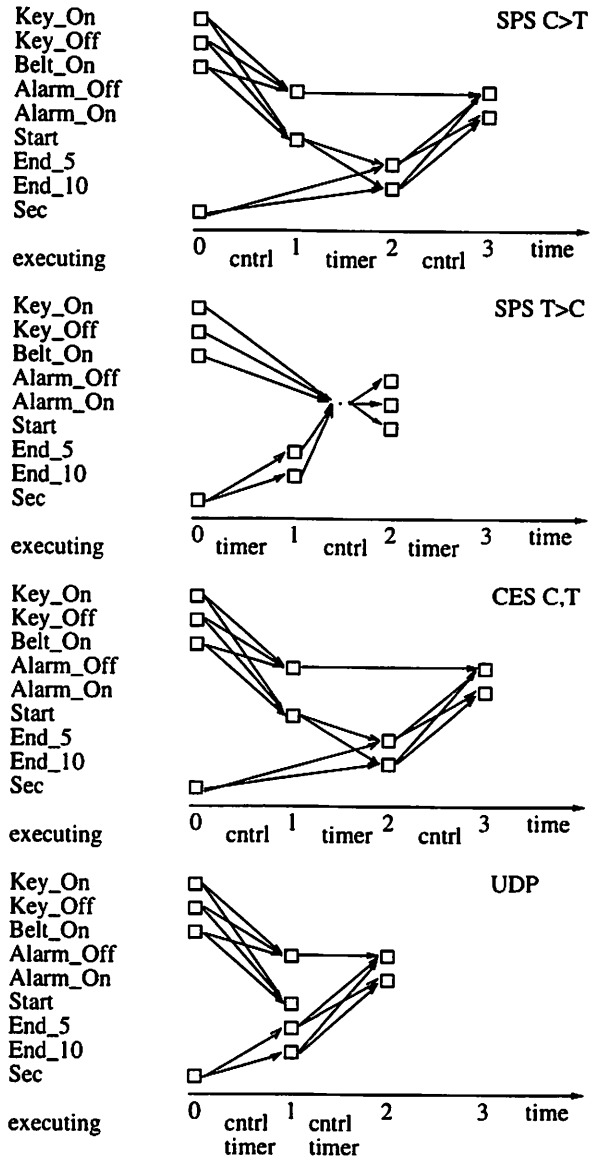

Figure 7.14: Least Execution Cover for Seat Belt Example.

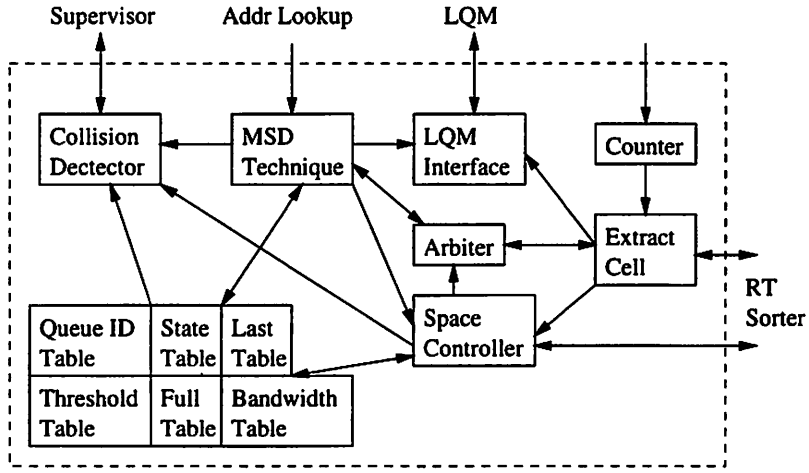

Figure 7.15: Algorithm Block of a ATM Server.

the cell is accepted, the Message Selective Discarding Technique sends instructions to the Logic Queue Manager about where (i.e. in which queue) to store the cell in the shared buffer. Communication between the algorithm block and the LQM is handled by the LQM interface, which performs the required protocol adaptations.

We used LEC communication analysis to decide synchronous equivalence among many different implementations, including ones with UDP and various SPS and CES scheduling policies. The LECs were generated using POLIS [BCG+97] on a network where CFSMs were replaced with their least non-decreasing covers. In all cases, the generation of the LEC took less than 1 second of CPU time. The LEC associated with the SPS policy chosen by an expert designer consists of 314 containers. Even with such a complex LEC, we have found a CES scheduling policy that is synchronously equivalent to the designer-specified SPS, therefore resulting in an implementation with less scheduling overhead.

We have performed LEC communication analysis for several different SPS, CES, and UDP scheduling policies. In each case, the LEC is finite even though there are many loops in this heavily interacting design. The analysis was performed in a very short computation time and with negligible memory occupation.

7.2.6 Correctness Proof of Theorem 7.5

We were able to (conservatively) check two implementations for synchronous equivalence by comparing their ECs, as implied by Theorem 7.5, that we repeat here for read-

ability.

Theorem 7.6 *If two implementations of a given* CFSM *network with well-behaved scheduling policies have identical ECs, then they are synchronously equivalent.*

We prove the theorem in several steps. Given two policies P and Q, we first construct auxiliary policies P^{EC} and Q^{EC}. The policy P^{EC} (Q^{EC}) has the same *Select* function as P (Q), but uses a different *Enable* function, based on the EC of P (Q respectively). Then, we prove that P and P^{EC} are synchronously equivalent (the same proof applies to equivalence of Q and Q^{EC}). Finally, we show that P^{EC} and Q^{EC} are equivalent, and the desired result follows by transitivity.

Lemma 7.7 *Consider two scheduling policies for an arbitrary* CFSM *network. If the two policies are well-behaved and have the same Select function, then they are synchronously equivalent.*

Proof: By definition, the execution traces generated by the two policies will be the same, except for empty executions. Therefore, the value of the last event emitted on some output will be the same for both policies. ∎

For some well-behaved scheduling policy P we construct a policy P^{EC}, by changing the *Enable* function. The rules for P^{EC} follow the outline of the EC-generating procedure, except that in Step 5, in addition to abstract execution, the actual selected CFSM is executed with actual inputs. Note that P and P^{EC} are synchronously equivalent by Lemma 7.7.

Lemma 7.8 *The following holds:*

- *when a* CFSM *is executed in Step 5 of the EC algorithm, any input with value* 1 *is also an active input (as determined in Step 2),*
- *if a* CFSM *emits an output in Step 5, then a corresponding container is also created by abstract execution.*

Proof: By induction, using the fact that the non-decreasing cover evaluates to x (presence of container) whenever the actual output function evaluates to 1. It will remain at x if some of its inputs are changed from 0 to x. ∎

In other words P^{EC} can be seen as determining the contents of the containers in the EC, 1 if the event is present and 0 if it is absent. Lemma 7.8 ensures that for every event that is actually generated, there exists a container to put it in.

Corollary 7.9 *The content of a container determined by P^{EC} depends only on the contents of its immediate predecessors in the EC, and the* CFSM *output function.*

Proof: By definition, all active inputs are immediate predecessors, and by Lemma 7.8, all other inputs are 0. ∎

The following result states that the EC contains sufficient information to decide synchronous equivalence.

Lemma 7.10 *If two scheduling policies P and Q have identical ECs and if P^{EC} and Q^{EC} assign to corresponding nodes the same contents for every primary input assignment, then P and Q are synchronously equivalent.*

Proof: We first show that P^{EC} and Q^{EC} are synchronously equivalent. Indeed, by definition of EC, all containers corresponding to some primary output form a chain, and the last container in the chain has the highest level (which may be different for P^{EC} and Q^{EC}). By assumption, the contents of the last containers are the same for P^{EC} and Q^{EC}, implying that P^{EC} and Q^{EC} are synchronously equivalent. It follows then by Lemma 7.7 that P and Q are also synchronously equivalent. ∎

Now we have all the pieces to prove the theorem.

Proof:[of Theorem 7.6] By Lemma 7.10 we only need to show that given two different well-behaved policies P and Q, their modifications P^{EC} and Q^{EC} assign the same values to corresponding containers. This is shown by induction, using Corollary 7.9, and the fact that the CFSM functions are the same in the two implementations. ∎

7.3 Conclusions

Communication analysis strives to answer the equivalence question by a single pass abstract execution of the implementations. We have shown how it can effectively analyze synchronous equivalence between different implementations. We also presented a detailed proof of the correctness of using execution covers as communication signatures in analyzing synchronous equivalence. The efficiency of communication analysis comes at

the cost of the conservativeness of the result, either in the form of simple false negatives or infinite execution covers, even when the actual implementation does not have infinite execution traces. By storing the presence/absence pattern of signals at each invocation of *Select* function of the scheduling policy, the execution cover generation procedure can detect when the abstract execution of an implementation is caught in an infinite loop. Practically, we generate execution covers for the two implementations in parallel. The analysis returns an inconclusive result if the execution covers differ at any point, or if either implementation is caught in an infinite loop. Otherwise, the two implementations are synchronously equivalent.

We still need to answer the question as to what happens when the result is inconclusive. We need a way to refine the analysis to make the execution cover finite, and to determine whether the inconclusive result is due to a true negative or a false negative. In the next chapter, we formally present a method to refine communication analysis. It can remove the false negatives and generate finite execution covers in more cases, at the cost of longer computation times. At the limit of this refinement process, the analysis becomes no longer abstract (i.e. it is an exact analysis).

Chapter 8

Refining Communication Analysis

Communication analysis is efficient in deciding the equivalence between two different implementations of the same high level specification. When two implementations have communication signatures (i.e. execution covers) that are identical and finite, they are guaranteed to be synchronously equivalent to each other. Unfortunately, when two execution covers are different, or when one or both of the execution covers are infinite, the result of the analysis is inconclusive. In the context of design exploration, the inconclusive result in equivalence analysis means that an implementation must be declared, possibly falsely, to be functionally different from the reference. If that implementation has a better performance characteristic than all the implementations that were declared to be functionally correct, it will still not be selected. If the negatives do turn out to be false, the final implementation is suboptimal.

In this chapter, we identify and remove, bit by bit, the sources of these false negatives through the process of refinement and pruning. By successive applications of refinement of containers and pruning of unobservable containers, we establish a smooth path to exhaustive simulation, where the analysis result is absolutely precise and no false negative is possible. When two implementations are actually synchronously equivalent to each other, communication analysis, perhaps with some refinement and pruning, can prove this positive result without going all the way down to exhaustive simulation. Figuring out precisely how much refinement and pruning are needed, and on which containers, is as hard as the verification problem itself. Heuristic algorithms can be developed, but that is left as possible topic of future research.

We deal with the issue of refinement in Section 8.1 and 8.2. In Section 8.1, we

discuss refinement on event containers. The containers in the previous chapter are all event containers. Refining an event container is the same as setting it to event presence, or event absence, and performing separate abstract simulations. It is not hard to see that if the corresponding "pre-split" EC portions, the "presence" EC portions, and the "absence" EC portions for the two implementations are all identical, they are synchronously equivalent to each other. The complication comes in that the non-decreasing cover functions have to take on an additional value, 1, which represents the event that is certainly present, as opposed to "possible" presence denoted by x. We prove that the ternary execution covers are also correct communication signatures.

The communication analysis of Chapter 7 abstracted away state information completely. Conservatism stemmed from ignoring the CFSM state need to be removed for more precise analysis. In Section 8.2, we represent state information as "state value" containers, one for each state value. One-hot encoding of state values allows easy representation of "sets of states". The various flavors of communication analysis of the previous chapter can be seen as representing state abstractly as a set containing all states. Refining the "set of all states" into subsets of states or individual states is similar to the container refinement. A separate abstract simulation is performed for each different case. It is not hard to see that if the corresponding "pre-split" portions and the separate refined portions of the two implementations are all identical, the implementations are synchronously equivalent to each other. Partitioning into subsets of states is similar in spirit to state space decomposition techniques [CHM+94], though with different goals. We want to partition the states to quickly show either equivalence or a counterexample.

In Section 8.3, we show that the local nature of communication analysis can be dealt with by pruning away unobservable containers from the execution cover. A container is unobservable if its content cannot affect, even transitively, the content of containers associated with primary outputs. Only the "observable" portions of the execution covers need to be compared for synchronous equivalence to hold.

With refinement and pruning, communication analysis can be made precise. We place communication analysis in the context of abstraction and refinement, and provide primitives, in the form of splitting and combining containers, to move around this abstraction/refinement spectrum. Using these primitives to devise heuristics or domain specific solutions is left as future work.

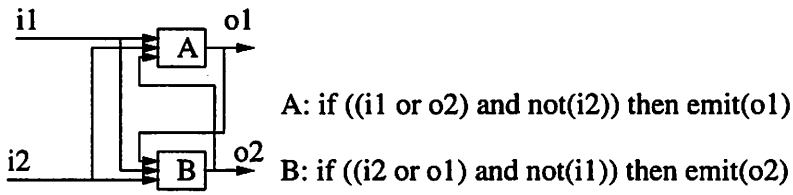

Figure 8.1: A Simple Example for Container Refinement.

8.1 Container Refinement

In communication analysis, containers are used to represent the possibility of an event. If there is no container at some input, it is interpreted as event absence. If there is a container present at some input, the event may be actually present, or it may not.

Consider the example in Figure 8.1. The output function is:

$$\begin{aligned} o_1 &= i_1 * \overline{i_2} + o_2 * \overline{i_2} \\ o_2 &= i_2 * \overline{i_1} + o_1 * \overline{i_1} \end{aligned} \tag{8.1}$$

The least non-decreasing cover for this function is:

$$\begin{aligned} \mathbf{o_1}^{\{x\}} &= \mathbf{i_1}^{\{x\}} + \mathbf{o_2}^{\{x\}} \\ \mathbf{o_2}^{\{x\}} &= \mathbf{i_2}^{\{x\}} + \mathbf{o_1}^{\{x\}} \end{aligned} \tag{8.2}$$

The LECs for two different implementations are shown in Figure 8.2. In both cases, the LEC are infinite and the communication analysis returns inconclusive result.

It can be established independently through exhaustive simulation not only that these two implementations produce finite output traces for any finite input trace, but also that they are actually synchronously equivalent to each other. The infinite LEC in Figure 8.2 is due to the abstraction of event containers. To make the abstract simulation finite, containers need to be "refined" to take on more precise information.

With the definition of container given in the previous chapter, there is no way to represent the precise presence of event, as it is possible to represent the precise absence of event by the absence of container at the input of a component. Execution covers in the previous chapter are computed with two values for each input variable of a component function: event absence, represented by 0, and event unknown, represented by x. For the analysis to be more precise, and less conservative, we need to represent the case where the event is definitely present.

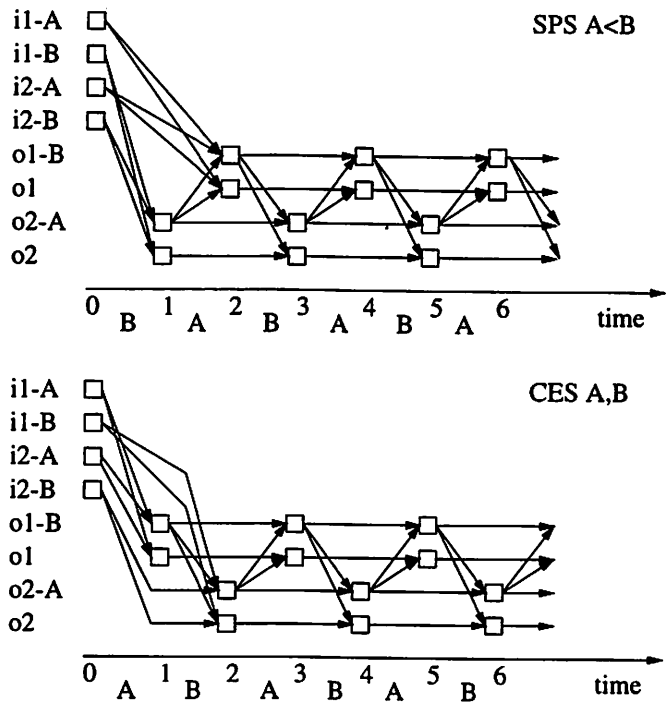

Figure 8.2: Least Execution Cover for Example of Figure 8.1.

Definition 8.1 (Ternary Container Set) *A* **ternary container set,** C, *is a set with three symbols:* 0, 1, *and* x. *I.e.* $\mathtt{C} = \{0, 1, \mathtt{x}\}$.

Definition 8.2 (Ternary Container-Event Set) *A* **ternary container-event set,** CE, *is a set with five symbols:* 0, 1, x, 0, *and* 1. *I.e.* $\mathtt{CE} = \{0, 1, \mathtt{x}, 0, 1\}$.

Notice that symbols in the container set and the event set are written with different fonts. If the reader finds it hard to distinguish a 1 in the container set from a 1 in the event set, he can rest assure that the distinction is only necessary in the strictest mathematical sense. Both of them mean that an event definitely has occurred. We can define n-dimensional spaces as follows:

Definition 8.3 (Ternary Container Space) *A n-dimensional ternary container space is defined as* $\mathtt{C}^n = \{0, 1, \mathtt{x}\}^n$.

Definition 8.4 (Ternary Container-Event Space) *A n-dimensional ternary container-event space is defined as* $\mathtt{CE}^n = \{0, 1, \mathtt{x}, 0, 1\}^n$.

0 in the ternary container space denotes the absence of the container, which corresponds to the absence (0) of event in event space. 1, or 1-container, in the ternary container space denotes the presence of event and the container, which corresponds to the presence (1) of the event in event space. x, or x-container, in the ternary container space denotes the presence of the container only, which corresponds to both the presence (1) and absence (0) of the event in the event space.

The information ordering in the ternary container set is naturally defined to be $0 \succeq x$, $1 \succeq x$. 0 and 1 both have at least as much information as x (i.e. 0 and 1 are at least as definite). 0 and 1 are incomparable in terms of information ordering. More formally, we extend the definition from the previous chapter.

Definition 8.5 ((Ternary) Information Ordering) *Ternary information ordering ($\succeq$) is defined as a relation between two symbols in the event set, ternary container set, and the ternary container-event set:*

- *Event set:*
  $0 \succeq 0; 1 \succeq 1;$
- *Ternary container set:*
  $0 \succeq 0; 1 \succeq 1; 0 \succeq x; 1 \succeq x; x \succeq x;$
- *Ternary container-event set:*
  $0 \succeq 0; 1 \succeq 1; 0 \succeq 0; 1 \succeq 1; 0 \succeq x; 1 \succeq x; x \succeq x; 0 \succeq 0; 0 \succeq x; 1 \succeq 1; 1 \succeq x; 0 \succeq 0; 1 \succeq 1.$

The information ordering can be extended to minterms in spaces by coordinate-wise extension. For example: $01 \succeq 01$, but $01 \not\succeq 00$; $x1 \succeq xx$, but $x0 \not\succeq 0x$; $01 \succeq 0x$, but $11 \not\succeq 01$.

The definition of non-decreasing functions and cover functions for the ternary container space are identical to those for the binary container space. We restate them here for convenience.

Definition 8.6 ((Ternary) Non-Decreasing) *Given two ternary container space minterms* $\mathbf{m_1}$ *and* $\mathbf{m_2}$*, a ternary container space function* $\mathbf{f}$ *is* **Non-Decreasing** *if and only if*

$$\mathbf{m_1} \succeq \mathbf{m_2} \Longrightarrow \mathbf{f}(\mathbf{m_1}) \succeq \mathbf{f}(\mathbf{m_2}) \tag{8.3}$$

A ternary container space function is **Non-Decreasing** if and only if changing any input from x to 0 or 1 will not change its value from 0 or 1 to x.

Definition 8.7 ((Ternary) Cover) *A ternary container space function* **f** *is a ternary cover of the event space function f if, under a given one to one correspondence between input variables.*

- *given any event space minterm m and any ternary container space minterm* **m**

$$m \succeq \mathbf{m} \Longrightarrow f(m) \succeq \mathbf{f(m)} \tag{8.4}$$

Intuitively, a ternary container space function **f** is a cover of the event space function f if, for any input minterm **m** in the container space,

- assigning 0 to an event space variable if its corresponding ternary container space variable has the value 0,
- assigning 1 to an event space variable if its corresponding ternary container space variable has the value 1,
- assigning 0 or 1 to an event space variable if its corresponding ternary container space variable has the value x,

all the resulting minterm m in the event space evaluate F to 1(0), **F** evaluates to 1(0) or x for **m**; otherwise, **F** evaluates to x for **m**.

Ternary execution covers for an implementation with a given scheduling policy can be obtained by "simulating", "abstractly executing" or "unfolding" the system graph by the following *Ternary Execution Cover Generation Procedure*:

1. Create containers for primary inputs and label them with level 0. Set the current level to 0. The input container may be filled, left with content unknown, or not be created at all, depending on what is known about the inputs. For example, if an input is known to be present for all cycles, then it can be set to be a 1-container.
2. Determine the set of *active* inputs for each (CFSM) node. A CFSM input is active if there exists a corresponding container with a level that is larger than that used by the previous execution level of that CFSM (i.e. there exists a container that has not been "consumed" by the previous execution).

3. Let all CFSMs with at least one active input be enabled. If no CFSM is enabled, then STOP.

4. Apply the *Select* function of the original scheduling policy to choose the CFSMs to be "executed".

5. *Abstractly execute* the selected CFSMs by increasing the current level by 1 and evaluating a ternary non-decreasing cover of each CFSM output function, with all active inputs with a 1-container set to 1, all active inputs with an x-container set to x, and all other inputs set to 0. If the ternary non-decreasing cover evaluates to 1 or x then:

   - create a new 1-container or x-container, respectively, and label it with the current level and the name of that output,
   - for every active input: create an edge from the most recent container corresponding to that input, to the newly created container,
   - create an edge from the previous container labeled with the same name (if any exists) to the new container.

6. "de-activate" the inputs of the executed CFSMs. Go to Step 2.

The ternary non-decreasing covers in Step 5 are computed by converting the output functions from event space to ternary container space. We present only the *least ternary non-decreasing cover*. We perform the following operations to obtain the least ternary cover function **F**. For every ternary minterm:

- The output of the ternary cover function is 1 if and only if, for all states, assigning any combination of 0 and 1 to the input x's produces 1 in the event space.
- The output of the ternary cover function is 0 if and only if, for all states, assigning any combination of 0 and 1 to the input x's all produces 0 in the event space.
- otherwise, the ternary cover function evaluates to x.

The ternary non-decreasing cover can be computed implicitly, or conservatively. Both topics are beyond the scope of this dissertation. The formulation of the ternary cover is reminiscent of symbolic simulation in the circuit domain [SB94]. The major difference is that symbolic simulation utilizes x to represent both 1 and 0 in a continuous waveform. An

interval at value x in the waveform means that the value of the signal can be 0 or 1 at any point of the interval. It says nothing about how many transitions (0 to 1 or 1 to 0) there may be. Our ternary container space is defined for discrete event systems. An event is defined as a 0 to 1 transition on the "wire" carrying the signal. Our x is used to denote the possibility of a single 0 to 1 transition. x in the container analysis represents the possibility of a single discrete event which, if it exists, can only be used (or consumed) once.

The ternary function calculated above is a cover of the corresponding event space function. It can output 0 only when all corresponding assignments in the event space never produce an output in any state. It can output 1 only when all corresponding assignments in the event space always produce an output for all states. The ternary cover function calculated above is non-decreasing. If the output is already 0 or 1, "resolving" any input that originally had the value x into 0 or 1 can only produce the same (0 or 1) output as before.

The ternary cover for the example in Figure 8.1 is:

$$
\begin{aligned}
o_1^{\{1\}} &= i_1^{\{1\}} * i_2^{\{0\}} + o_2^{\{1\}} * i_2^{\{0\}} \\
o_1^{\{0\}} &= i_2^{\{1\}} + i_1^{\{0\}} * o_2^{\{0\}} \\
o_1^{\{x\}} &= \overline{o_1^{\{1\}} + o_1^{\{0\}}} \\
o_2^{\{1\}} &= i_2^{\{1\}} * i_1^{\{0\}} + o_1^{\{1\}} * i_1^{\{0\}} \\
o_2^{\{0\}} &= i_1^{\{1\}} + i_2^{\{0\}} * o_1^{\{0\}} \\
o_2^{\{x\}} &= \overline{o_2^{\{1\}} + o_2^{\{0\}}}
\end{aligned}
\tag{8.5}
$$

Computing the ternary execution cover with primary inputs set to x at Step 1 produces an infinite execution cover exactly like that in Figure 8.2. This is expected because we have not really done any refinement on the containers at all. We only changed the cover functions so that they can handle the 1-containers. Abstract execution with i_1 refined at the primary input produces the execution covers shown in Figure 8.3. We now have a "split" execution cover. One, on the left side of the figure, with i_1=1, denoted by a solid box. The other, on the right side of the figure, with i_1=0, denoted by no box at i_1. If the corresponding portions of the execution covers are identical, we can say that the two implementations are synchronously equivalent to each other, since i_1 can only be 0 or 1 in the original event space.

Unfortunately, the refined LEC is still infinite. Further refinement is needed. The

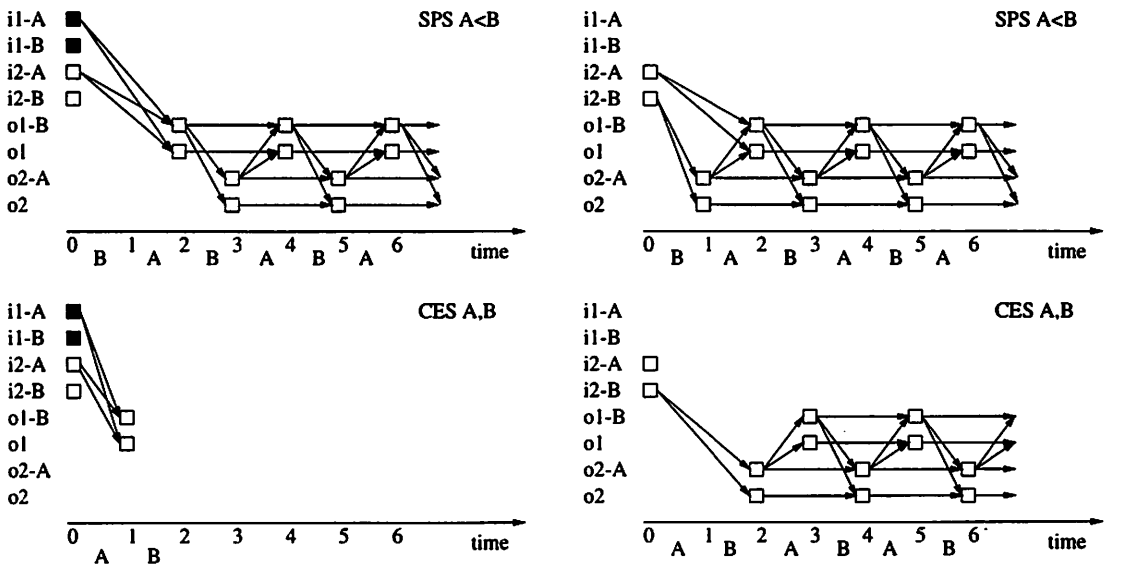

Figure 8.3: Refined LEC for Example in Figure 8.1.

new execution covers is shown in Figure 8.4. After both $\mathbf{i_1}$ and $\mathbf{i_2}$ input containers are refined, the corresponding LECs are indeed identical. The implementations are therefore synchronously equivalent to each other. We do not have to consider the case where both $\mathbf{i_1}$ and $\mathbf{i_2}$ are refined to 0, since the system is reactive by definition. The absence of all inputs cannot cause any reaction.

We are able to (conservatively) check two implementations for synchronous equivalence by comparing their ternary ECs, as stated by the following result.

Theorem 8.1 *If two implementations of a given* CFSM *network with well-behaved scheduling policies have identical ternary ECs, then they are synchronously equivalent.*

The theorem and its proof are ternary extensions of the ones in Section 7.2.6. We prove the theorem in several steps. Given two policies P and Q, we first construct auxiliary policies P^{EC} and Q^{EC}. The policy P^{EC} (Q^{EC}) has the same *Select* function as P (Q), but uses a different *Enable* function, based on the ternary EC of P (Q respectively). Then, we prove that P and P^{EC} are synchronously equivalent (the same proof applies to equivalence of Q and Q^{EC}). Finally, we show that P^{EC} and Q^{EC} are equivalent, and the desired result follows by transitivity.

For some well-behaved scheduling policy P we construct a policy P^{EC}, by changing the *Enable* function. The rules for P^{EC} follow the outline of the ternary EC-generating

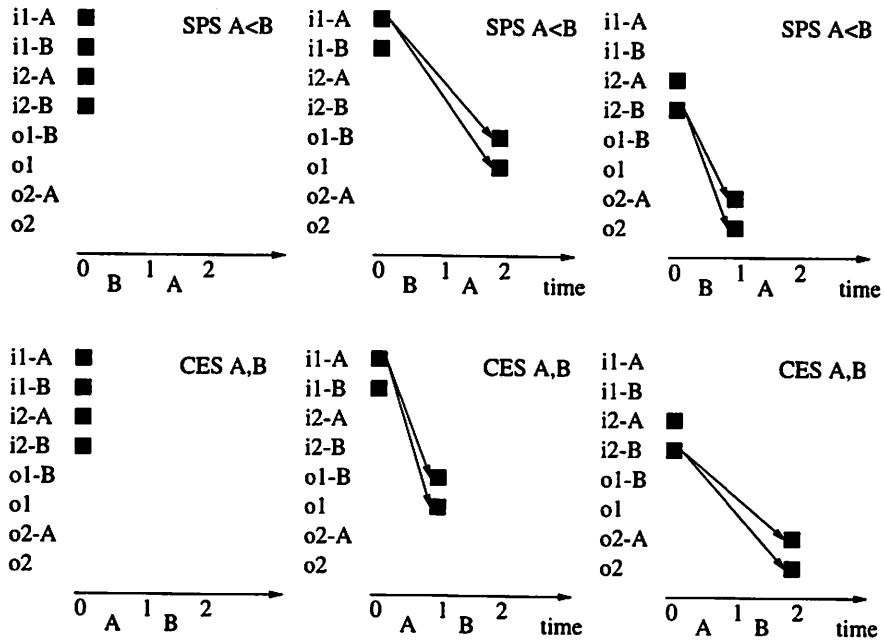

Figure 8.4: Further Refined LEC for Examples in Figure 8.1.

procedure, except that in Step 5, in addition to abstract execution, the actual selected CFSM is executed with actual inputs. Note that P and P^{EC} are synchronously equivalent by Lemma 7.7.

Lemma 8.2 *The following holds:*

- *when a* CFSM *is executed in Step 5 of the ternary EC algorithm in this chapter, any input with value* 1 *is also an active input (as determined in Step 2),*
- *if a* CFSM *emits an output in Step 5, then a corresponding container,* 1 *or* x, *is also created by abstract execution.*

Proof: By induction, using the fact that the non-decreasing ternary cover evaluates to x or 1 whenever the actual output function evaluates to a event space 1. If the non-decreasing ternary cover evaluates to x, then it will remain x even if some inputs are changed from 0 or 1 to x. If the non-decreasing ternary cover evaluates to 1, then it may either remain 1 or become x when some input is changed from 0 or 1 to x. In either case, the container is still present. ∎

In other words P^{EC} can be seen as determining the contents of the x containers in the ternary EC, 1 if the event is present and 0 if it is absent. Lemma 8.2 ensures that for every event that is actually generated, there exists a container to put it in.

Corollary 8.3 *The content of a container determined by P^{EC} depends only on the contents of its immediate predecessors in the ternary EC, and on the* CFSM *output function.*

Proof: By definition, all active inputs are immediate predecessors, and by Lemma 8.2, all other inputs are 0. ∎

The following result states that the ternary EC contains sufficient information to decide synchronous equivalence.

Lemma 8.4 *If two scheduling policies P and Q have identical ternary ECs and if P^{EC} and Q^{EC} assign to corresponding nodes the same contents for every primary input assignment, then P and Q are synchronously equivalent.*

Proof: We first show that P^{EC} and Q^{EC} are synchronously equivalent. Indeed, by definition of ternary EC, all containers corresponding to some primary output form a chain, and the last container in the chain has the highest level (which may be different for P^{EC} and Q^{EC}). By assumption, the contents of the last containers are the same for P^{EC} and Q^{EC}, implying that P^{EC} and Q^{EC} are synchronously equivalent. It follows then by Lemma 7.7 that P and Q are also synchronously equivalent. ∎

Now we have all the pieces to prove the theorem.

Proof:[of Theorem 8.1] By Lemma 8.4 we only need to show that given two different well-behaved policies P and Q, their modifications P^{EC} and Q^{EC} assign the same values to corresponding containers. This is shown by induction, using Corollary 8.3, and the fact that the CFSM functions are the same in the two implementations. ∎

The refinement does not have to be limited to the primary input containers (as is the case of Figure 8.4). At any point in the ternary execution cover generation procedure, we can decide to refine any x-container and generate a "split" execution cover from that point. One side of the split corresponds to placing a 1-container at that point, the other side of the split corresponds to placing a 0-container at that point, or no container at all. The refined ternary execution cover is still a valid communication signature, as demonstrated by the following theorem:

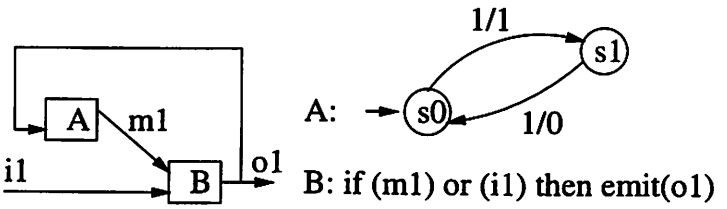

Figure 8.5: A Simple Example for State Refinement.

Theorem 8.5 *If two implementations of a given* CFSM *network with well-behaved scheduling policies have identical refined ternary ECs, then they are synchronously equivalent.*

Proof: By Lemma 8.4 we only need to show that given two different well-behaved policies P and Q, their modifications P^{EC} and Q^{EC} assign the same values to corresponding containers. This is shown by induction, using Corollary 8.3, the fact that the container refinement procedure assigns the same values to corresponding "split" portions of the execution cover, and the fact that the CFSM functions are the same in the two implementations. ∎

8.2 State Refinement

For some design, the false negatives from communication analysis cannot be removed without explicitly representing states. Consider the example in Figure 8.5. The ternary least non-decreasing covers for the functions are:

$$
\begin{aligned}
\mathbf{m_1}^{\{1\}} &= 0 \\
\mathbf{m_1}^{\{0\}} &= \mathbf{o_1}^{\{0\}} \\
\mathbf{m_1}^{\{x\}} &= \mathbf{o_1}^{\{x\}} + \mathbf{o_1}^{\{1\}} \\
\mathbf{o_1}^{\{1\}} &= \mathbf{m_1}^{\{1\}} + \mathbf{i_1}^{\{1\}} \\
\mathbf{o_1}^{\{0\}} &= \mathbf{m_1}^{\{0\}} * \mathbf{i_1}^{\{0\}} \\
\mathbf{o_1}^{\{x\}} &= \mathbf{m_1}^{\{x\}} * \mathbf{i_1}^{\{0\}} + \mathbf{m_1}^{\{0\}} * \mathbf{i_1}^{\{x\}} + \mathbf{m_1}^{\{x\}} * \mathbf{i_1}^{\{x\}}
\end{aligned}
\tag{8.6}
$$

The ternary LECs for two selected scheduling policies are shown in Figure 8.6. One cannot conclude whether or not the two implementations are synchronously equivalent because the LECs are infinite. Even though the infinite LECs "look" similar, one cannot

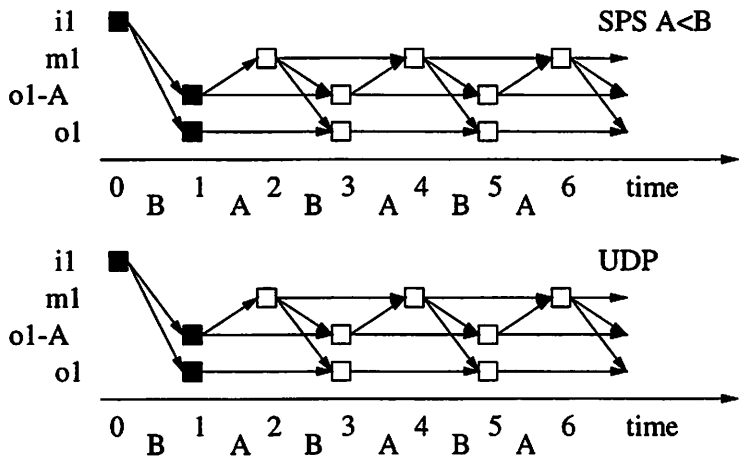

Figure 8.6: LEC for Example in Figure 8.5.

draw any conclusion about the finite execution traces from the infinite execution covers. The infinite execution cover may "block" the analysis algorithm from analyzing the entire execution trace. It can also be shown that no amount of event container refinement can make the execution cover finite. The infinite execution cover is caused by the abstraction of state values.

To represent state information, we encode state in one-hot fashion much like we did for the valued events. State values have the following property.

Lemma 8.6 *A set of one-hot encoded state value signals for a* CFSM *has a* 1*-container if and only if it is the only container present for that set of state values.*

Proof: By definition, any CFSM must be in some state at any given time. Only one container present means that there is only one state that it can "possibly" be in. The "event" is therefore definitely present. Conversely, if there is a 1-container at some state value, then the CFSM can not possibly be in any other state, so other state values must not have any container present. ∎

For the sake of brevity and clarity, we will convert state value signals with a single x-container immediately into a 1-container. Minor modifications need to be made for the ternary execution cover generation procedure to handle state value containers differently. After converting both value event and state values into "events" with one-hot encoding, the *Modified Ternary Execution Cover Generation Procedure* is:

1. Create containers for primary inputs and state values of the CFSMs and label them with level 0. Set the current level to 0. The input containers may be filled, left with content unknown, or not be created at all, depending on what is known about the inputs.

2. Determine the set of *active* inputs for each (CFSM) node. A CFSM input is active if there exists a corresponding container with a level that is larger than that used by the previous execution level of that CFSM (i.e. there exists a container that has not been "consumed" by the previous execution).

3. Let all CFSMs with at least one non-state active input be enabled. If no CFSM is enabled, then STOP.

4. Apply the *Select* function of the original scheduling policy to choose the CFSMs to be "executed".

5. *Abstractly execute* the selected CFSMs by increasing the current level by 1 and evaluating a ternary non-decreasing cover of each CFSM output function, with all active inputs with a 1-container set to 1, all active inputs with an x-container set to x, and all other inputs set to 0. If the ternary non-decreasing cover evaluates to 1 or x then:

   - create a new 1-container or x-container, respectively, and label it with the current level and the name of that output,
   - for every active input: create an edge from the most recent container corresponding to that input, to the newly created container,
   - create an edge from the previous container labeled with the same name (if any exists) to the new container.
   - for state value containers, additionally create an edge from all previous state value containers to all new state value containers.

6. "de-activate" the inputs of the executed CFSMs. Go to Step 2.

The ternary non-decreasing cover in Step 5 is computed by converting the output function from event space to the ternary container space, taking into consideration the state values. There are many possible covers, but we only present the *modified least ternary non-decreasing cover*. We perform the following operations to obtain the *modified least ternary cover* function **F**. For every ternary minterm:

- The output of the ternary cover function is 1 if and only if all event space minterms obtained by assigning any combination of 0 and 1 to the input x's produce that output (1) in the event space. At any given time, one and only one state value container is assigned 1 while other x-containers for that state are assigned 0.
- The output of the ternary function is 0 if and only if all event space minterms obtained by assigning any combination of 0 and 1 to the input x's produce no output (0) in the event space. At any given time, one and only one state value container is assigned 1 while other x-containers for that state are assigned 0.
- otherwise, the ternary function evaluates to x.

The modify ternary non-decreasing cover for component A in the example in Figure 8.5 becomes:

$$\begin{aligned}
\mathbf{m_1}^{\{1\}} &= \mathbf{o_1}^{\{1\}} * (\mathbf{s_0}^{\{1\}} + \mathbf{s_1}^{\{0\}}) \\
\mathbf{m_1}^{\{0\}} &= \mathbf{o_1}^{\{0\}} + \mathbf{s_0}^{\{0\}} + \mathbf{s_1}^{\{1\}} \\
\mathbf{m_1}^{\{x\}} &= \overline{\mathbf{m_1}^{\{1\}} + \mathbf{m_1}^{\{0\}}} \\
\mathbf{Next_s_0}^{\{1\}} &= \mathbf{o_1}^{\{1\}} * (\mathbf{s_0}^{\{0\}} + \mathbf{s_1}^{\{1\}}) + \mathbf{o_1}^{\{0\}} * (\mathbf{s_0}^{\{1\}} + \mathbf{s_1}^{\{0\}}) \\
\mathbf{Next_s_1}^{\{1\}} &= \mathbf{o_1}^{\{1\}} * (\mathbf{s_0}^{\{1\}} + \mathbf{s_1}^{\{0\}}) + \mathbf{o_1}^{\{0\}} * (\mathbf{s_0}^{\{0\}} + \mathbf{s_1}^{\{1\}}) \\
\mathbf{Next_s_0}^{\{0\}} &= \mathbf{Next_s_1}^{\{1\}} \\
\mathbf{Next_s_1}^{\{0\}} &= \mathbf{Next_s_0}^{\{1\}} \\
\mathbf{Next_s_0}^{\{x\}} &= \overline{\mathbf{Next_s_0}^{\{1\}} + \mathbf{Next_s_0}^{\{0\}}} \\
\mathbf{Next_s_1}^{\{x\}} &= \overline{\mathbf{Next_s_1}^{\{1\}} + \mathbf{Next_s_1}^{\{0\}}}
\end{aligned} \tag{8.7}$$

(8.8)

From the fact that there are only two state values, Lemma 8.6 explains why $\mathbf{s_0} = 1$ whenever $\mathbf{s_1} = 0$, and why $\mathbf{s_0} = 0$ whenever $\mathbf{s_1} = 1$. Component B has the same cover function as before:

$$\begin{aligned}
\mathbf{o_1}^{\{1\}} &= \mathbf{m_1}^{\{1\}} + \mathbf{i_1}^{\{1\}} \\
\mathbf{o_1}^{\{0\}} &= \mathbf{m_1}^{\{0\}} * \mathbf{i_1}^{\{0\}} \\
\mathbf{o_1}^{\{x\}} &= \mathbf{m_1}^{\{x\}} * \mathbf{i_1}^{\{0\}} + \mathbf{m_1}^{\{0\}} * \mathbf{i_1}^{\{x\}} + \mathbf{m_1}^{\{x\}} * \mathbf{i_1}^{\{x\}}
\end{aligned} \tag{8.9}$$

With the new ternary non-decreasing cover, which takes state values into consideration, we are able to "refine" also the state value containers and "split" the execution cover.

In Figure 8.7, the LECs on the left-hand side correspond to computation phases where component A starts at state s_0. The right-hand side corresponds to computation phases where component A starts at state s_1. The state in which a component "starts" during a computation phase does not have anything to do with the initial state of that component. Potentially, a component can start a computation phase in any one of its reachable states. It is not hard to see that, after splitting the execution covers according to the starting states, if the corresponding execution covers are identical, then the two implementations are synchronously equivalent to each other.

We do not have to refine state value containers into individual states all at once. x-containers at multiple state value signals represents a subset of states. The procedure we outlined can already take subsets of states into consideration. The state refinement also does not have to be limited to the beginning of the generation procedure. At any point in the modified ternary execution cover generation procedure, we can decide to refine any state value container and generate a "split" execution cover from that point. One side of the split corresponds to a particular subset of states, while the other side of the split corresponds to the rest of the states. The refined ternary execution cover is still a valid communication signature, as demonstrated by the following theorem:

Theorem 8.7 *If two implementations of a given* CFSM *network with well-behaved scheduling policies have identical refined modified ternary ECs, then they are synchronously equivalent.*

Proof: By Lemma 8.4 we only need to show that given two different well-behaved policies P and Q, their modifications P^{EC} and Q^{EC} assign the same values to corresponding containers. This is shown by induction, using Corollary 8.3, the fact that the refinement procedure assigns the same values to corresponding "splits" of the execution cover, and the fact that the CFSM functions are the same in the two implementations. In particular, splitting the execution cover for state refinement assigns the same "subset of states" for each corresponding portion of the execution cover. ∎

8.3 Pruning Execution Covers

Synchronous equivalence, as defined in Chapter 5, relates to the primary inputs and outputs of a design. Communication analysis infers this global property from the

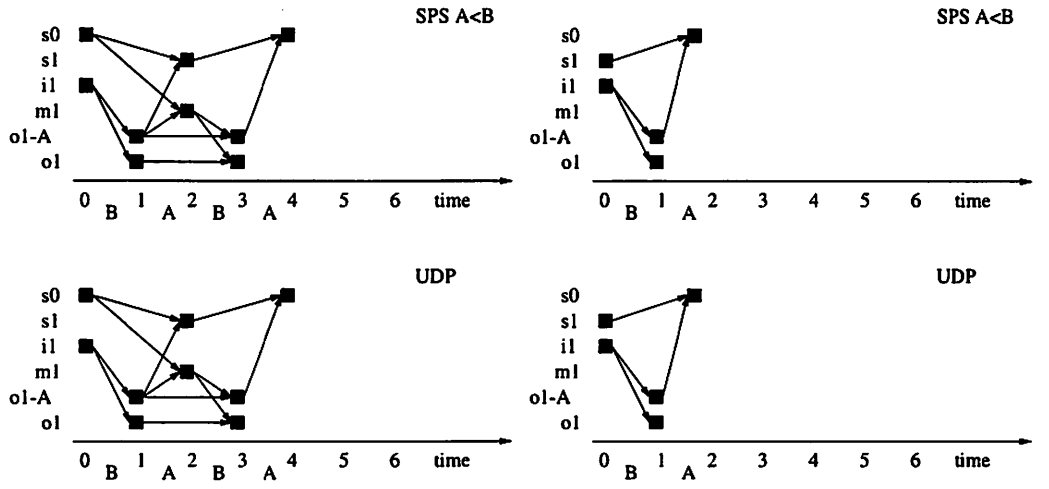

Figure 8.7: Refined LEC for Example in Figure 8.5.

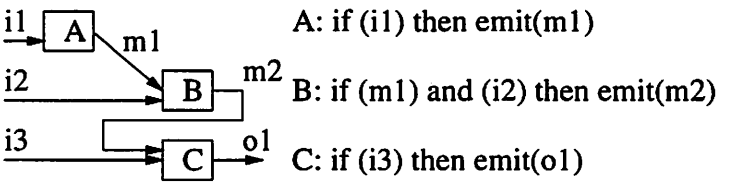

Figure 8.8: An Example Demonstrates the Need for Pruning.

communication between components, which is not a strictly global characteristic. Inferring a global property from characteristics of local elements can be conservative. Consider the example in Figure 8.8. LECs for three different scheduling policies are shown in Figure 8.9. It can be established through exhaustive analysis, or simple observation, that the three implementations are actually synchronously equivalent to each other, though there are two different sets of LECs. It is equally apparent that many containers in the LECs have nothing to do with the primary output o_1 and are indeed *unobservable* from the primary outputs.

Definition 8.8 (Observable Container) *A container c in an execution cover is observable if there is an assignment of* x*-containers, consistent with transition and output relations, such that assigning* 0 *or* 1 *to c produces different primary outputs for the actual implementation. Otherwise, c is unobservable.*

The next lemma clarifies the relationship between an implementation producing a primary output, and the primary output containers in an execution cover.

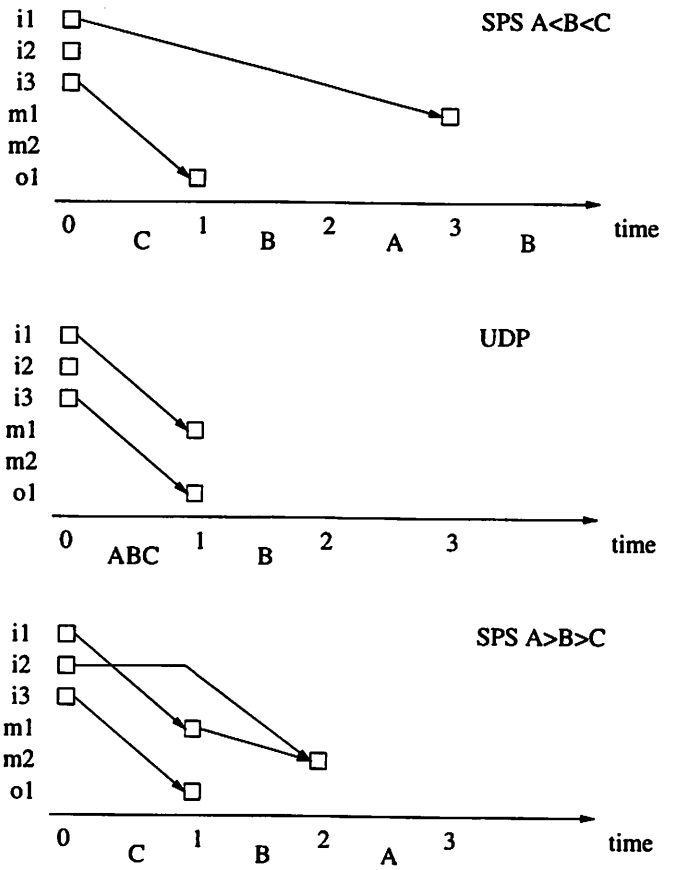

Figure 8.9: Least Execution Covers for Example in Figure 8.8.

Lemma 8.8 *If there are one or more* 1*-containers at some primary output of an execution cover, the output is emitted by the corresponding implementation.*

Proof: By the definition of synchronous equivalence (Definition 5.2), primary outputs matter only at the end of a cycle. By the definition of CFSM, an emitted output cannot be "cancelled". ∎

A container that is not observable can be removed from the execution cover.

Theorem 8.9 *Removing unobservable containers, along with all their edges, results in an execution cover that remains a communication signature of the implementation.*

Proof: Consider an implementation, A, its execution cover, C, and an execution cover, C^{Pruned}, where some unobservable containers have been removed. Let A^{Pruned} be an implementation that will produce the execution cover C^{Pruned}. By the definition of unobservable

containers and synchronous equivalence, A^{Pruned} is synchronously equivalent to A. By transitivity of equivalence relations, any implementation that is synchronously equivalent to A^{Pruned} is also synchronously equivalent to implementation A. ■

Removing all the unobservable containers from an execution cover is analogous to the classical observability problem in sequential circuits [Wan96]. Finding an exact solution could require an exponential computation, though simple pruning can go a long way in removing a large number of these unobservable containers.

Corollary 8.10 *If there are one or more* 1*-containers at some primary output of an execution cover, all but one of the* 1*-containers of that primary output can be removed and the execution cover remains a communication signature of the implementation.*

Proof: By Lemma 8.8, the output will be emitted if one or more 1-containers exist at some primary output of the execution cover. This will remain true if all but one of these 1 containers are removed. By Theorem 8.9, the pruned execution cover is a communication signature of the implementation. ■

Corollary 8.11 *Any non-primary-output container that has no directed edge to an* x*-container can be removed and the execution cover remains a communication signature of the implementation.*

Proof: Assigning 1 or 0 to a container with no directed edge to an x-container clearly cannot affect the primary output, so the container is unobservable and can be removed. By Theorem 8.9, the pruned execution cover is a communication signature of the implementation. ■

A large number of unobservable containers can be removed by the following procedure, and according to Corollary 8.10 and Corollary 8.11, the pruned execution cover is a communication signature of the implementation.

1. For all primary outputs that have one or more 1-container, remove all containers of that primary output except a single 1-container.

2. Remove all non-primary-output containers that do not have an edge to an x-container. If no such container can be found, STOP.

3. Return to Step 2.

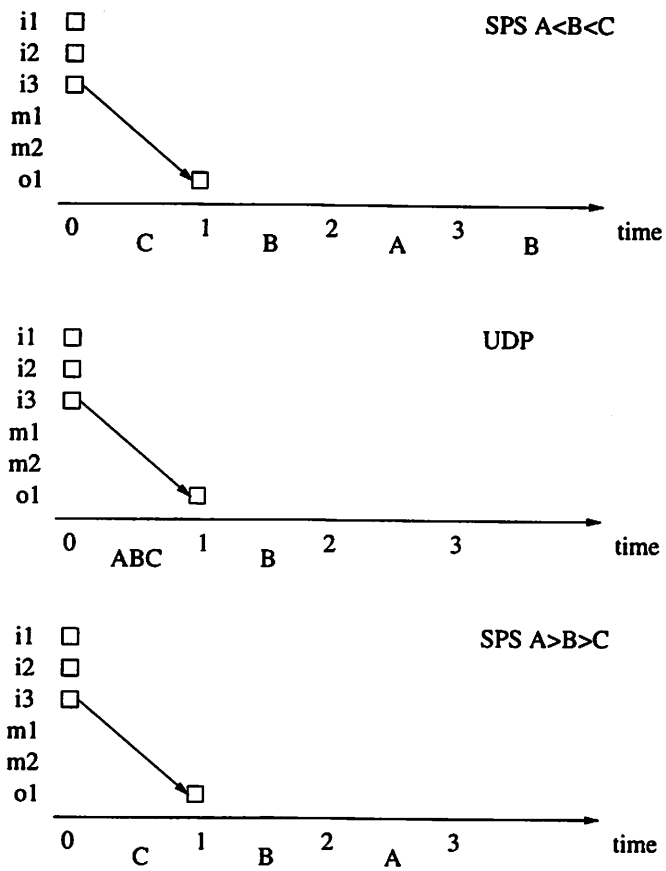

Figure 8.10: Pruned Least Execution Covers for Example in Figure 8.8.

The pruned execution covers for the example in Figure 8.8 (derived by pruning the original execution cover in Figure 8.9) are shown in Figure 8.10. The original primary input containers are retained for clarity. The implementations are indeed synchronously equivalent to each other.

8.4 Relationship with Exhaustive Simulation

With container refinement and pruning, execution covers can be related to exhaustive simulation. Exhaustive simulation requires a deterministic transition and output relation, as well as a deterministic input trace. A deterministic input trace for an actual implementation corresponds to an "abstract" execution with an input trace consisting of only 1-containers.

Theorem 8.12 *Given a deterministic transition relation and output relation, a well-behaved scheduling policy, and deterministic input trace, the output trace from simulation is identical to a trace of the pruned execution cover from the corresponding abstract simulation of the cover functions.*

Proof: The transition and output relations are deterministic, and the primary input of the abstract simulation consists of only 1-containers. By Step 5 of the modified ternary EC generation algorithm, only 1-containers will exist in the execution cover. By Corollary 8.10 and Corollary 8.11, only a single 1-container at each primary output will remain. The trace of the execution cover is therefore the same as the output trace of the corresponding simulation run. ∎

The theorem relates a simulation run of the implementation to an abstract execution of its non-decreasing covers. Not all refined pruned execution covers have corresponding simulation runs, since not all combinations of the states of the components are reachable, and the given pruning algorithm does not identify all unobservable containers. Both of these problems can be solved with existing state reachability techniques either exactly or heuristically.

Chapter 9

Conclusions and Future Directions

In this chapter we summarize our contributions and point out some directions where further research work can proceed.

9.1 Conclusions

The goal of this dissertation was to investigate how formal methods can be applied to the domain of embedded system design. The emphasis was on the specification, representation, validation, and design exploration of such systems from a high level perspective. We started by reviewing the framework and formal model upon which the theories and experiments are based, and in which the formal methods that we developed are linked to synthesis and simulation. We then took on the issues of formal verification and abstract equivalence checking.

We formulated a formal verification methodology to verify general properties of CFSM networks. We demonstrated that this methodology is efficient in dealing with the problem of complexity and effective in finding bugs. However, manual intervention is required, coming in the form of assumption and abstraction selection. Assumptions are made in the form of separate timing and functionality verifications through constraints on the architectural mappings. The application of formal verification techniques to CFSM networks led us to the following conclusion. For a formal abstract methodology to be effective, it must be applied to a specific property, not just any arbitrary property. The property must also allow for the separation of concerns. In specific, it must allow for the separate specifications and validations of timing and functionality.

Therefore, we focused on the abstract analysis of the synchronous equivalence of two implementations under the synchronous assumption. This property is analogous to functional equivalence for sequential circuits and takes full advantage of the separation of timing and functionality. One powerful result of this equivalence criterion is the identification of a set of delay insensitive scheduling policies. Once a delay insensitive scheduling policy is chosen, any variation in delay does not affect the functional behavior. Static analysis of this sort is very efficient and effective in proving equivalence, though it may at times be conservative. To check for equivalence between implementations using different delay insensitive scheduling policies, we proposed algorithms based on worst-case analysis of the communication among components. The events communicated between components are abstracted into a signature that is maximal in the sense that it represents all possible communication patterns of that implementation. By comparing the signatures of different delay insensitive scheduling policies for a given system, we were able to determine equivalence conservatively. We demonstrated with real-life examples that synchronous equivalence opens design exploration avenues uncharted before.

Lastly, we related communication analysis to exhaustive simulation through a series of refinement and pruning operations on the communication signatures. An algorithm can choose to work at any abstraction level trading off computational efficiency with the possibility of inconclusive result due to false negatives. We provided primitives to move amongst different abstraction levels that exist between abstract communication analysis and exhaustive simulation.

9.2 Future Directions

Much research work is still needed before the synchronous approach can be as popular in the embedded system domain as it is in the sequential circuit domain. There are three different research directions that may prove to be crucial in this regard. They are

- Iterative refinement techniques.
- Repartitioning of the components.
- Multi-clock operation.

9.2.1 Iterative Refinement Techniques

During communication analysis, if two communication signatures differ, or if either of them is infinite, the result of the analysis is inconclusive. The "error trace" of the analysis is simply the difference in execution covers. Based on the error trace, containers can be refined to make the analysis more precise. This can be accomplished manually, with the primitives suggested in Chapter 8. It could also be done through automatic techniques similar to those proposed in [Bal94] for formal verification of timed systems.

The goal of the iterative refinement will be to reach a positive result as early as possible, or to reach a single branch (split) of completely refined execution covers where for some primary output, one implementation produces a 1 while the other produces a 0. These two objectives are contradictory and user hints such as "probably equivalent" or "probably not equivalent" can help in choosing the refinement that can more quickly lead to a solution.

9.2.2 Component Repartitioning

An interesting open question is what happens if we violate the assumption that the component functionalities and the component connectivity are the same among all possible implementations. General repartitioning can be thought of as a series of decompositions and compositions of the CFSMs. It is not hard to compare two implementations where the difference is only that one component is decomposed into two or two components are combined into one through synchronous decomposition and composition. If the "subnetwork" represented by the two decomposed components, for the given implementation, satisfies the synchronous assumption, then it is equivalent to the implementation with a single, synchronously composed component. How this check can be done efficiently and locally will be crucial for an efficient general repartitioning methodology.

9.2.3 Multi-Clock Synchronous Systems

It is straightforward to apply abstract communication analysis to acyclic systems with two different hardware resources running at two different "clocks", since a complete ordering of the component executions can be found which has a period equal to the least common multiplier of the two periods. In fact, we can show that one can always construct a single process CES implementation that is synchronously equivalent to the multi-clock synchronous hardware implementation of the same acyclic specification. This principle can

also be extended to synchronized processors running at different "macro clocks". However, it is not clear that under what condition can this "synthesis" approach to design exploration be extended to cyclic networks.

Bibliography

[ABD+95] Neil C. Audsley, Alan Burns, Robert I. Davis, Ken W. Tindell, and Andy J. Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Systems*, 8(2-3):173–198, 1995.

[Bal94] Felice Balarin. *Iterative Methods for Formal Verification of Discrete Event Systems.* PhD thesis, University of California Berkeley, 1994.

[Bal99] F. Balarin. Worst-case analysis of discrete systems. In *Proceedings of the International Conference on Computer-Aided Design*, November 1999.

[BCG91] G. Berry, P. Couronné, and G. Gonthier. The synchronous approach to reactive and real-time systems. *IEEE Proceedings*, 79, September 1991.

[BCG+97] Felice Balarin, Massiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Claudio Passerone, Alberto Sangovanni-Vincentelli, Ellen Sentovich, Kei Suzuki, and Bassam Tabbara. *Hardware-software co-design of embedded systems: the Polis approach.* Kluwer Academic Publishers, Boston; Dordrecht, 1997.

[BCL+94] J. Burch, E. Clarke, D. Long, K. McMillan, et al. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design*, 13(4):401–424, April 1994.

[Ber96] Gérard Berry, 1996. See http://cma.cma.fr/Esterel.

[BHLM90] J. Buck, S. Ha, E.A. Lee, and D.G. Masserschmitt. Ptolemy: a framework for simulating and prototyping heterogen eous systems. *Interntional Journal of Computer Simulation*, special issue on Simulation Software Development, January 1990.

[BRB90] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient Implementation of a BDD Package. In *Proc. of* 27^{th} *Design Automation Conference*, pages 40–45, June 1990.

[BSV97] F. Balarin and A. Sangiovanni-Vincentelli. Schedule validation for embedded reactive real-time systems. In *Proceedings of the Design Automation Conference*, June 1997.

[BSVA+96] R. Brayton, A. Sangiovanni-Vincentelli, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, S. Qadeer, R. Ranjan, T. Shiple, G. Swamy, T. Villa, G. Hachtel, F. Somenzi, A. Pardo, and S. Sarwary. VIS: A System for Verification and Synthesis. In *Proc. of the 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer-Verlag, 1996.

[Buc93] J. T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model.* PhD thesis, U.C. Berkeley, 1993. UCB/ERL Memo M93/69.

[BY93] Felice Balarin and Gary York. Verilog HDL modeling styles for formal verification. In *Proceedings of the IFIP Conference on Hardware Description Languages and their Applications*, April 1993.

[CDV94] Coppo, M. D'Ambrosio, and V. Vercellone. The A-VPN server, a solution for atm virtual private networks. *Proceedings of ICCS*, November 1994.

[CGH+99] M. Chiodo, P. Guisto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, E. Sentovich, and K. Suzuki. Synthesis of software programs for embedded control applications. *IEEE Transactions on Computer-Aided Design*, 18(6):834–849, June 1999.

[CHM+94] H. Cho, G. Hachtel, E. Macii, M. Poncino, and F. Somenzi. A structural approach to state space decomposition for approximate reachability analysis. In *Proc. of Int'l Conference on Computer Design*, October 1994.

[Den75] J. B. Dennis. First version data flow procedure language. Technical Report MAC TM61, Massachusetts Institute of Technology, May 1975.

[DH89] D. Drusinski and D. Har'el. Using statecharts for hardware description and synthesis. *IEEE Transactions on Computer-Aided Design*, 8(7), July 1989.

[Eng94] Institute of Electrical and Electronics Engineers. *IEEE standard VHDL language reference manual.* IEEE, 1994.

[FLL+98] E. Filippi, L. Lavagno, L. Licciardi, A. Montanaro, M. Paolini, R. Passerone, M. Sgroi, and A. Sangiovanni-Vincentelli. Intellectual property re-use in embedded system co-design: an industrial case study. *International Symposium on System Synthesis*, December 1998.

[GL94] O. Grumberg and D. Long. Model checking and modular verification. *IEEE Transactions on Programming languages and Systems*, 16(3), May 1994.

[GM92] M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, 1992.

[GR94] D. Gajski and L. Ramachandran. Introduction to high-level synthesis. *IEEE Design and Test of Computers*, 11(4):44–54, 1994.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HLN+90] D. Har'el, H. Lachover, A. Naamad, A. Pnueli, et al. STATEMATE: a working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4), April 1990.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, languages and Computation.* Addison Wesley, 1979.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of IFIP Congress*, August 1974.

[Kur94] R. P. Kurshan. *Automata-Theoretic Verification of Coordinating Processes.* Princeton University Press, 1994.

[LL73] C.L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46 – 61, January 1973.

[LLSV99] M. Lajolo, M. Lazarescu, and A. Sangiovanni-Vincentelli. A compilation-based software estimation scheme for hardware/software co-simulation. In *Proceedings of the International Workshop on Hardware-Software Codesign*, May 1999.

[LM87] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *IEEE Proceedings*, September 1987.

[MBNSV93] A. Malik, R. Brayton, A. Newton, and A. Sangiovanni-Vincentelli. Two-level minimization of multivalued functions with large offsets. *IEEE Transactions on Computers*, 42(11), November 1993.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[NPW81] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains. part I. *Theoretical Computer Science*, 13:85–108, 1981.

[ORR+96] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In *Proc. of the 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer-Verlag, 1996.

[SAB+93] T. Shiple, A. Aziz, F. Balarin, S. Cheng, R. Hojati, T. Kam, S. Krishnan, V. Singhal, H. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Formal design verification of digital systems. In *Proceedings of TECHCON*, 1993.

[SB94] Carl Seger and Janusz Brzozowski. Generalized ternary simulation of sequential circuits. *Informatique Theorique et Applications*, 28(3–4):159–86, 1994.

[SBT96] T. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *Proceedings of European Design and Test Conference*, March 1996.

[SSL+92] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, U.C. Berkeley, May 1992.

[Tho90] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science.* Elsevier, 1990.

[TSL+90] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines Using BDD's. In *Proc. of Int'l Conference on Computer-Aided Design*, November 1990.

[Ung69] S. H. Unger. *Asynchronous Sequential Switching Circuits.* Wiley Interscience, 1969.

[Wan96] Huey-Yih Wang. *Hierarchical Sequential Synthesis: Logic Synthesis of FSM Networks.* PhD thesis, University of California Berkeley, 1996.

[YMS+98] J. Young, J. MacDonald, M. Shilman, A. Tabbara, P. Hilfinger, and A. Newton. Design and Specification of embedded Systems in Java Using Successive, Formal Refinement. In *Proc. of* 35^{th} *Design Automation Conference*, pages 70–75, June 1998.

Index