ABSTRACT

Title of Dissertation: AUTOMATIC ANALYSIS OF CONSISTENCY BETWEEN REQUIREMENTS AND DESIGNS

Maria "Marsha" A. Chechik, Doctor of Philosophy, 1996

Dissertation directed by: Professor John Gannon
Department of Computer Science

Writing requirements in a formal notation permits automatic assessment of such properties as ambiguity, consistency, and completeness. However, verifying that the properties expressed in requirements are preserved in other software life cycle artifacts remains difficult.

This dissertation describes methods and tools for automatically analyzing the consistency of software requirements and detailed designs. Requirements describe systems as state machines with event-driven transitions using a specification language that is easy to read and scalable to large systems. We have defined consistency between requirements and detailed designs to be that exactly the same transitions and global safety properties specified in the requirements are preserved in the designs. To judge global properties like these, we determine the possible system states which exist at different points in designs. The detailed bookkeeping necessary to do this exceeds the capabilities of human reviewers for all but small systems.

We have developed a language for specifying detailed designs, an analysis technique to create a model of a design through abstract interpretation of the language constructs, and a method to automatically generate and check properties derived from requirements to ensure a design's consistency with them. These ideas are implemented in a tool which is capable of analyzing designs with multiple procedures and processes. Using our tool, we were able to uncover errors in designs of existing sequential systems and analyze our own designs of concurrent systems.

AUTOMATIC ANALYSIS OF CONSISTENCY BETWEEN REQUIREMENTS AND DESIGNS

by

Maria "Marsha" A. Chechik

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland at College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
1996

Advisory Committee:

Professor John Gannon, Chairman/Advisor
Associate Professor Richard Gerber
Associate Professor William Pugh
Professor Alan Robock, Dean's Representative
Professor Rouben Rostamian, Math Dept., UMBC

DEDICATION

To my parents, Arkady and Nina Chechik.

ACKNOWLEDGEMENTS

I was truly blessed by being surrounded by extremely intelligent and extremely supportive people. I would have never done it without them.

Fist of all, I am very grateful to my thesis advisor, John Gannon, who was always there for me. Discussions with him lead to many ideas described here. And he patiently read through many many drafts of this thesis. However, his most important contribution was his firm faith in me, even when I gave no reasons for it.

I would also like to thank Bill Pugh, who read this thesis several times, carefully and thoroughly. He found a number of fundamental errors in the original document, and made many of helpful technical suggestions. Thanks to him, this document is (more) correct and complete. Although at times I felt like a first-grader who forgot to do her homework, working with Bill was a great pleasure.

Rich Gerber deserves a special mention. Throughout the years, he has played a very important role in my life, being an informal mentor and a friend, who encouraged me along and whose door was always open, whether I needed a pad on the back or an advise on my research. He has taught me model-checking and, more importantly, gave me a framework to solve research problems.

Working with Connie Heitmeyer and Stuart Faulk at Naval Research Lab made me understand the difficulty and importance of trying to specify and verify real software systems. They also taught me to go for the bottom line, rather than attempt to cover all cases. Connie and Stuart also made many technical suggestions about my work.

Rouben Rostamian, my first Calculus teacher, talked me into taking up mathematics as the second major in college. He and many other professors of the Math Department at UMBC, gave me so much encouragement and support while I was an undergraduate, that I had enough arrogance and momentum to go through all the initial hurtles of graduate school.

This research was supported by the Air Force Office of Scientific Research under contract F49620-93-1-0034.

Finally, I would like to thank my family and friends for their support and understanding.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

AUTOMATIC ANALYSIS OF CONSISTENCY
BETWEEN REQUIREMENTS AND DESIGNS

Maria "Marsha" A. Chechik

December 2, 1996

This comment page is not part of the dissertation.

Typeset by LaTeX using the `dissertation` style by Pablo A. Straub, University of Maryland.

Chapter 1

Introduction

Precise documentation of software requirements has several potential benefits: designers know what they are to build; reviewers can check that customers' intentions are met; testers can formulate test cases independently from the system's implementation; and maintainers can use the original requirements to learn about the system before making their changes[Par93]. Although useful for specifying a variety of types of computer systems, precise documentation is essential for safety-critical systems[BS93].

1.1 Safety-Critical Systems

Safety-critical systems are "systems where compelling evidence is required that the system delivers its services in manner that satisfies certain critical properties"[HBGL95]. These systems must prevent unintended events that could result in "death, injury, occupational illness, damage to or loss of equipment or property, or environmental harm"[Lev86, oD84]. These systems must be designed so that they seldom if ever reach hazardous states, and if they do, they can gracefully recover from them.

More and more safety-critical systems are controlled by software. For example, most modern commercial aircrafts (since 1982) have automated flight management systems capable of controlling all aspects of the aircraft's flight plan, including takeoff and landing. The Therac 25 is a software-controlled medical instrument used in radiation therapy, capable of emitting both X-rays and beta rays. The Darlington nuclear plant has a shutdown cooling system controlled by software. Although systems such as telephone networks and financial systems are not normally classified as safety-critical, the consequences of failures of such systems can be very serious. If someone is critically ill and the telephone network fails to establish a connection to medical personnel on time, the person may die. Similarly, the failure of a bank to receive a large electronic transfer on time may result in huge financial losses.

Because the failure of such systems to satisfy their requirements can have catastrophic consequences, customers and developers are willing to invest considerable time, effort, and money in ensuring that such systems will not fail. Application of "formal methods" in development of such systems is a promising technique.

1.2 Formal Methods

Generally, the term formal method refers to the use of mathematical techniques in the design and analysis of computer hardware and software. These methods may use formal specification languages to describe the required system behavior and mathematically-based analysis techniques to demonstrate that system behaviors satisfy critical properties.

Some of the arguments for using formal methods are:

- The process of formalizing specifications forces the writer to state all assumptions, which helps to avoid omissions and expose inconsistencies often found in natural language descriptions.

- Formal methods allow properties of a computer system to be predicted from a mathematical model of the system. For example, logic theory can be used to determine whether a specification comprised of a set of logic formulas is satisfiable and consistent.

- Formal methods can be used to prove consistency of different software (or hardware) artifacts. For example, one can verify that a formal design specification is a refinement of a formal requirement specification.

Most software errors are introduced during the requirements phase[Off92]. Since there is a growing evidence that requirements errors can lead to serious accidents, and that fixing requirements errors can be extremely costly when done late in the software life-cycle, techniques for improving the quality of requirements documents are crucial. Formal methods play a big role in this effort. A number of formal requirements notations (e.g. the Software Cost Reduction (SCR) notation[AFB+88, Hen80] and the Requirements State Machine Language (RSML)[JLHM91]) have been used to write requirements for realistic applications and analyze their completeness and consistency[HBGL95, RPRL96, HL95].

Verifying that designs and implementations preserve the "correctness" of requirements is not easy. Usually, this is achieved via testing, which can provide only partial coverage of the range of behaviors of the system. The behavior of software is determined by a succession of discrete state changes. Test cases provide information on only the state sequences actually examined. Without an assumption of continuity (i.e., that if two "close" tests cases behave correctly, then all intermediate test cases will also behave correctly) there is little reason to suppose the behavior of untested sequences will succeed if the test cases succeed[Rus93]. Formal methods can be used to calculate properties of systems for all legal inputs.

Formal methods, however, are not frequently used on industrial applications. Applying formal methods requires a level of mathematical sophistication beyond that possessed by many software developers. Many formal methods lack adequate tool support. To be useful in developing practical systems, formal methods must increase software quality, produce readable documents, scale to large applications, keep development costs under control and be supported by tools[HM96]:

- **Readability**. A formal requirements document should be understood by users of the system, who usually have little technical background. One way to enhance readability is to "hide" precise mathematical notation under a layer of more intuitive graphical and/or tabular notations and to add explanatory comments.

- **Scalability**. Verification techniques have been criticized[Par85] for not being able to analyze anything but toy systems. A skilled designer can create a good abstraction of a system so that the analysis is tractable but the results obtained also hold in the actual system.

- **Reasonable costs**. Formal methods should be applied to creation of requirements documents to maximize their leverage on the software development process. The cost of creating formal descriptions of requirements can then be amortized across several different analysis activities.

- **Tool support**. Well-engineered tools not only reduce the cost of application of formal methods, but also increase the reliability with which they can be applied.

1.3 Current Trends in Formal Methods

1.3.1 Notations

Formal notations can be divided into three major classes: models based on logic, state-machines, and process algebras.

Logic-based models

Mathematical logic has been used to specify and prove properties about programs [Hoa69, Flo67]. Using logic, one can describe the behavior of the system without building it first, as well as specify requirements completely and precisely. Most real-time logics describe systems in terms of events, i.e., points of time when something significant occurs. Examples of such systems include Real-Time Logic or RTL[JM86], TRIO[GMM90] and ASTRAL[CPKM94]. RTL has been further extended (to RTTL) to describe temporal distance[Ost89b].

Several temporal logics have been developed to describe real-time systems. Linear-time semantics (e.g., [Pnu77, MP90, MP92]) describe particular executions of a system, whereas branching-time semantics (e.g., [EH86, CES86]) describe all possible executions. Interval-time semantics (e.g., [SMS82, All83]) are used to express conditions that hold for some nonzero time interval. UNITY[CM88] is a framework for specification and verification that uses extended state-transition systems to model abstract programs in a similar logic.

State-machine models

Statecharts[Har87] and its commercial version STATEMATE[HLN$^+$90] exploit the nature and simplicity of the classical finite state machine, enhancing it with hierarchical and compositional features. A set of associated tools performs reachability analysis and simulation. [GF88] use methods similar to Statecharts. However, the real-time aspects of Statecharts semantics are underdeveloped[MLJ88]. A number of other notations, based on Statecharts, address this problem. Both Modecharts[JM86] and TTM[Ost89b] use similar representations, allowing expression of timing and logical properties expressed in RTL and RTTL, respectively. Both approaches are supported with a suite of tools, including verifiers based on model-theoretic reasoning.

Petri nets[Pet81, Rei85] are used to reason about execution of concurrent processes with a number of (limited) resources. A Petri net is an operational formalism, supporting a notion of state and its evolution. Specifications expressed in this notation are executable, and verification is based on building a reachability graph (as in the Berthomieu-Diaz algorithm [BD91]). Two extensions, time Petri nets[MS76] and timed Petri nets (TPNs) [Ram74] were introduced for modeling time. Levenson and Stolzy[LS87] use TPNs for safety analysis, without having to generate the entire reachability graph, by working backward from hazardous states to determine if these states are reachable. Critical states (i.e., immediate predecessors of hazardous states) are identified and interlocks are added to these states to force the system to take only those paths that do not lead to hazardous states. This approach is called fault tree analysis. A similar backward approach is used in the real-time temporal logic approach to controller design[Ost89a, Ost91].

The systems described above are graphical notations, with formal underlying semantics. Other systems are described by textual notations. The Timed Automaton Model[LT88, LV91, MMT91] extends state machines with a notion of dense time. This model has an infinite number of states, where time is added to the usual state machine model via a special "time passage" action. Estrel[BdS91, BG92] is one of the family of languages used to describe real-time systems using the state machine model. These languages have been applied to a number of industrial projects, especially in the fields of nuclear plant safety and avionics[LRD95].

Process algebras

Process algebras were developed to specify and analyze concurrent systems. The original versions, such as CCS[Mil89], CSP[Hoa85] and ACP[BK85], were untimed. Each process algebra has a set of well-defined components: a concise language, a precise semantics, a notion of equivalence, and a set of algebraic laws allowing syntactic manipulation.

- The language is based on a small set of operators and a set of rules on how to combine the operators to describe complex systems.
- The semantics describes the possible execution steps a process can take.
- Two processes are <u>equivalent</u> if they exhibit the same behavior.
- Two descriptions of a system are written: a specification is described as an <u>abstract</u> process and an implementation is described as a <u>detailed</u> process.
- To prove correctness, two processes are shown to be equivalent to each other.

This technique is called <u>bisimulation</u>.

Process algebras able to express timing properties include: CSR[Ger91, GL92] and ACSR[PGJI93], which adds time to CCS; Timed CSP[RR87], an extension of CSP;

Timed LOTOS[BB87], an extension of the ISO standard LOTOS; and a timed version of ACP[JJ91].

1.3.2 Analysis Techniques

Common analysis techniques can be divided into three broad classes: model-theoretic reasoning, proof-theoretic reasoning, and simulation.

Model-checking

Model checkers[CES86] take a finite state machine model of a system and a temporal logic formula and determine if the formula holds in the model. The advantage of this approach is that the analysis is fully-automatic and does not require a user to understand sophisticated mathematical techniques like induction. Unfortunately, models of real systems have very large numbers of states. A number of approaches have been developed to handle the "state explosion problem". Unreachable and impossible states are pruned[CG87], large sets of states are described symbolically[BCM+90], data-structures like Binary Decision Diagrams (BDDs)[Bry86, Bry92] are used to store and analyze complex boolean formulae efficiently, and techniques are created to analyze subsets of the model and then combine the results of the analysis[CLM89]. Current model checkers can verify some systems with 10^{130} states [BCM+90]. Model checkers have been most effectively applied to verification of hardware and protocol specifications. For example, a Symbolic Logic Verifier (SMV) [McM93] was used to detect several errors in the Futurebus+ Cache Coherency Protocol, an IEEE standard.

A number of model-theoretic approaches incorporated analysis of real-time properties. Examples of such systems are TRIO[GMM90, FM91] and the Modechart verifier

[JS88]. In another approach[YMW93], SMV was extended to verify larger systems, by incorporating an efficient technique for timing and event counting.

Theorem-proving

Proof-theoretic reasoning uses logical rules of deduction to formally prove that a specification satisfies a property of interest or that one system specification is equivalent to another system description. These techniques are applied manually (with some computer support), and require users to have considerable mathematical skills and training.

Proof-theoretic techniques can handle more abstract models and produce more general results than model-checking. Deductive reasoning can be applied to any model, not necessarily a finite-state machine. A number of mechanical proof systems have been developed: the Boyer-Moore theorem prover[BM88], EVES[KPS+93], and the Larch Prover(LP) [GHW85, GH93b] are based on first-order logic; and HOL [Gor88] and PVS[OSR93] are based on higher-order logic. These systems do some proofs automatically, although most non-trivial proofs require user guidance. Mechanical proof systems can be very useful in checking hand-proofs, though.

Simulation

Simulation[CHLR93] or specification testing[Kem85] is a form of testing applied to formal requirements. Users experiment with the system specification by providing a set of inputs to the system, and a simulator uses the specification to show how the state of the system changes based on the input. This technique allows the user to ensure that the specification captures the intended system behavior. Unfortunately, evaluating software requirements via execution has the same drawbacks as testing; namely, the number of test cases needed to cover a specification is unmanageably large for all but the smallest systems[BF91].

1.4 Industrial Applications of Formal Methods

Formal methods can contribute to the development of systems in two ways[Rus93]:

1. they can improve the development process, leading to a better product, and/or reduced time and cost; or
2. they can contribute to the assurance and certification of a system.

Several industrial projects successfully used formal methods[CGR95]. For example, SCR methods have been applied to A-7 Operational Flight Control Program[Hen80], a submarine communication system[HM83] and safety-critical components of the Darlington nuclear power plant in Canada[vPM93]. The SACEM project used Hoare-style proofs to verify a one-line commuter train network in Paris[GH90] . The Traffic Alert and Collision Avoidance System (TCAS II) project used the Requirements State Machine Language [JLHM91] to express the system's requirements. IBM used Z in the development of a major new release of its transaction processing system CICS[HK91]. A semiconductor manufacturer Inmos used formal methods in the development of a floating point unit for the T800 Transputer[Bar89]. A formal specification and verification effort[MSS82] was associated with the Software Implemented Fault Tolerance (SIFT) project [ea78], aimed at development of a fault-tolerant computer for fly-by-wire passenger aircraft. LOTOS languages and tools were used to specify and model software for the Flight Warning Computer (FWC) of the Airbus 330 and 340 aircraft[GH93a].

In a more recent project, model-checking techniques have been applied to verify the t900 Virtual Channel Processor[Bar95].

1.5 Overview of Dissertation

Software quality can be improved by eliminating errors arising from inconsistencies within the description of a system or between two different descriptions of a system. Automated techniques have been used to derive a finite-state representation of a set of tabular requirements, and determine if it is a model for system safety properties expressed as temporal logic assertions[AG93]. This dissertation presents a complementary technique which automatically compares properties derived from a set of requirements with a finite state representation of a system's detailed design.

Designs are often expressed in a program design language (PDL)[CG75] consisting of a concrete outer syntax of basic statement types, and an inner syntax of comments. Design reviews seek to establish that the design is consistent with the requirements[Lam88]. We define a design to be consistent with its requirements if the design's state transitions are enabled by the same events as those of the requirements, all the requirement's state transitions appear in the design, and the requirement's safety properties hold in the design. In a large project, these properties may be checked by design and code inspections conducted by human reviewers. This process may be effective in catching local inconsistencies, but the bookkeeping tasks needed to determine all the possible system states at a particular program point make it difficult to ensure that global properties hold. In this paper, we define a notation which can be used as a PDL and describe a prototype tool, called Analyzer[CG94, CG95], which automatically determines if a design written in PDL is consistent with its requirements.

The inputs to our tool are a tabular requirements specification and a detailed design written in PDL. The PDL's outer syntax is comprised of C control constructs, and its inner syntax is a set of special comments which describe local properties of values of requirements variables. As Figure 1.1 illustrates, the tool generates a set of logic formulas corresponding to our notion of consistency between a design and its requirements and a finite-state machine FSM abstraction of the design. A special-purpose model checking algorithm determines if the FSM is a model of the formulas.

The rest of the dissertation is organized as follows:

- Chapter 2 describes the formal model of SCR requirements and our notion of consistency between SCR requirements and designs.

- Chapter 3 presents the language for creating detailed program designs and describes the process of building a finite-state machine from the detailed design.

- Chapter 4 describes the model-checking approach to check automatically-generated and user-defined system properties.

- Chapter 5 presents a case-study during which we analyzed an existing detailed design of a water-level monitoring system with respect to its SCR requirements.

- Chapter 6 discusses changes we made to the SCR requirements model, to the design language, and to the analysis technique to process designs of concurrent systems.

- Chapter 7 concludes this dissertation with a discussion of the successes and limitations of our analysis technique.

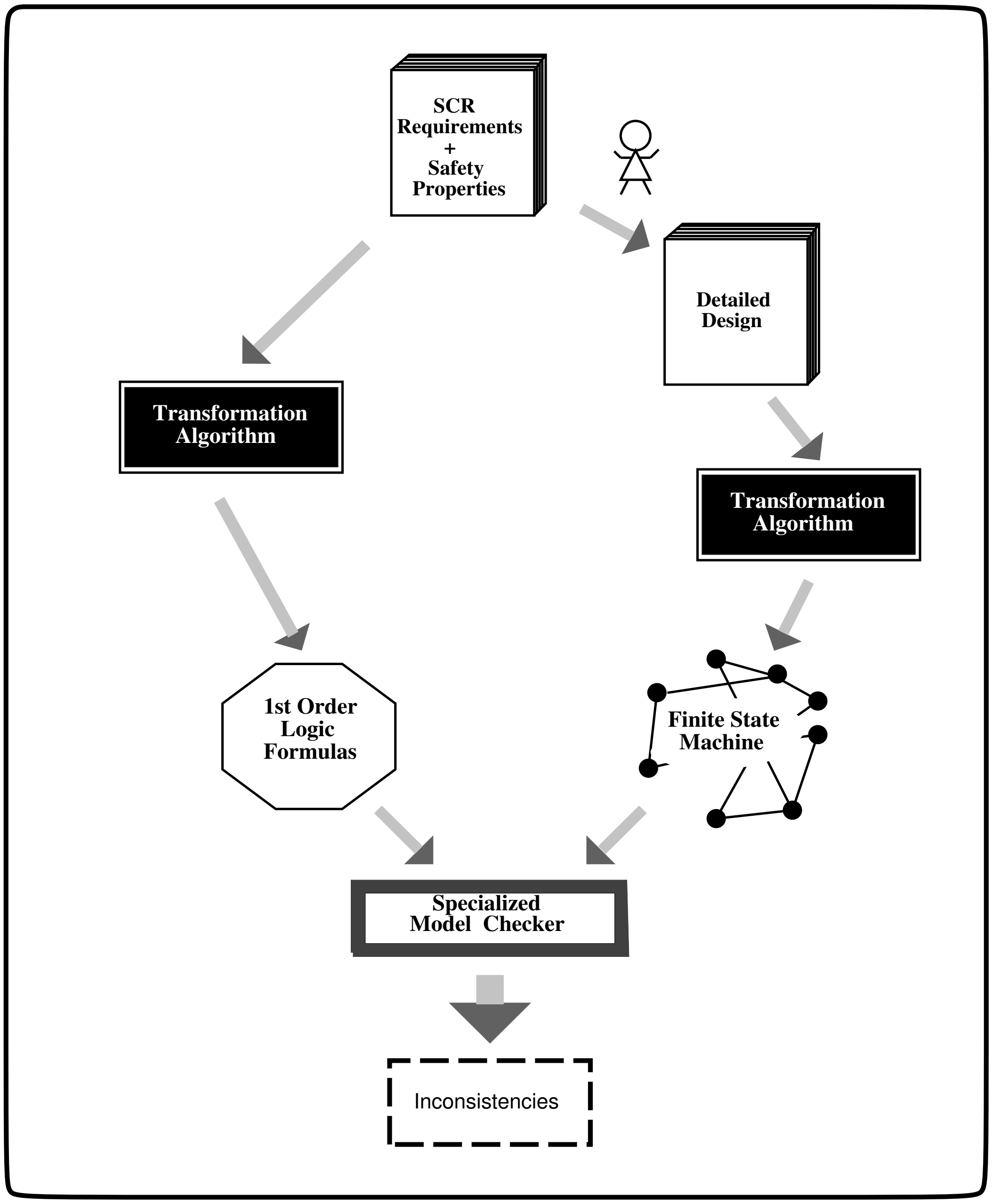


Figure 1.1: Algorithm to solve the problem.

Chapter 2

SCR Software Requirements

A software requirements document is usually the first textual description of a system, specifying the system's required behavior. Such a specification might represent a set of legal and illegal executions of the system or specify a set of properties which must hold in the system. Another way to represent requirements is to describe how control and data should flow among components of the system.

The SCR requirements notation was developed by a research group at the Naval Research Laboratory as part of the Software Cost Reduction project [AFB+88, Hen80]. The project resulted in SCR requirements and design standards, as well as guidelines for software development using SCR. A complete SCR requirements specification contains behavioral, functional, precision, and timing requirements of a software system as well as assumptions about the environment in which the system will operate. The specification language is precise, can be understood by engineers and software developers, and is easy to use and modify. The initial language lacked an underlying formal semantics. A number of semantics have been proposed[Atl92, HLK95, Fau89, PM91, van90], some of which[HLK95, PM95] became bases of tools automating consistency and completeness checks[HL93] and simulations[CHLR93, PW94] of requirements. The requirements may also be used to generate test oracles for testing program artifacts derived from requirements[PP94].

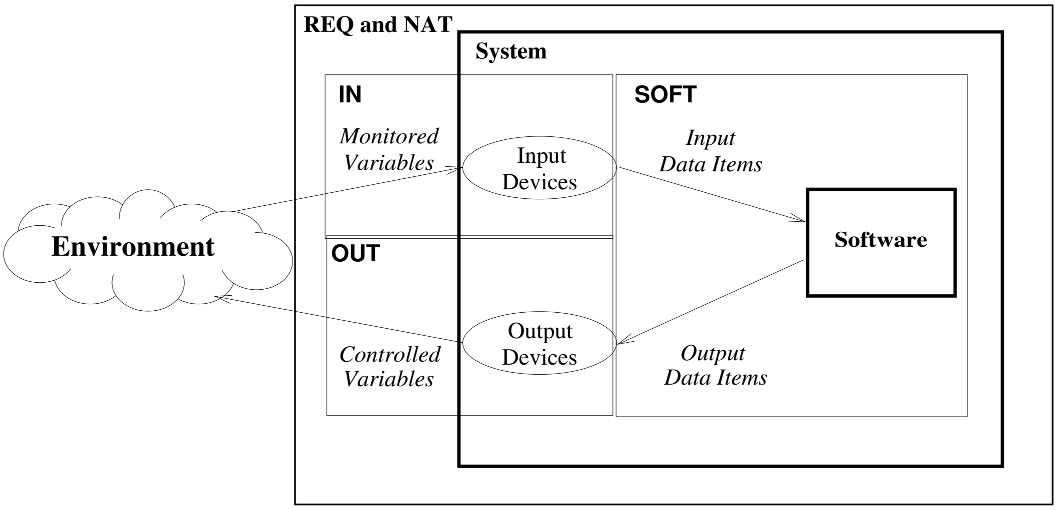


Figure 2.1: Four Variable Model.

In this chapter, we describe SCR behavioral requirements and environmental assumptions, and the system goals they are meant to ensure.

2.1 Four Variable Model

The Four Variable Model[PM91, PM95] describes required system functions, timing, and accuracy as a set of mathematical relations on four sets of variables: monitored and controlled variables, and input and output data items. Figure 2.1 illustrates the approach. A monitored variable represents an environmental system behavior, and a controlled variable represents an environmental quantity that the system controls. The specification of required system behavior is given as a black box, with monitored and controlled variables as its inputs and outputs, respectively. This specification is captured in two relations, REQ and NAT, from monitored to controlled variables.

- NAT defines a set of possible values of variables, describing the natural constraints on the system behavior. These constraints are imposed by laws of nature and by the system's environment.

- REQ defines additional constraints as relations between the monitored and the controlled variables.

In this model, input devices (i.e., sensors) measure the monitored variables, and output devices (i.e., actuators) set the controlled variables.

- The relation IN is a mapping from monitored variables to the input data items.
- OUT is a mapping between output data items and the controlled variables.
- Finally, SOFT is a mapping between input and output data items.

An SCR requirements specification can describe both the system and the software requirements by describing REQ and SOFT, respectively. The rest of this dissertation is concerned only with software requirements. Thus we use the term <u>monitored variables</u> to describe monitored variables or input data items, and <u>controlled variables</u> to describe controlled variables or output data items.

The SCR model is based on event-driven state-transition systems. We assume that time is discrete (because a sensor samples data periodically). Changes to monitored variables may cause the system to change its state or to alter the values of its controlled variables.

Definition 2.1 *SCR requirements* $\mathcal{R}$ *is a tuple* $\mathcal{R} = \langle \mathcal{B}, \mathcal{E}, \mathcal{G} \rangle$*, where*

$\mathcal{B}$ *is a set of behavioral requirements,*

$\mathcal{E}$ *is a set of environmental assumptions; and*

$\mathcal{G}$ *is a set of system goals.*

2.2 Behavioral Requirements

2.2.1 Conditions and events

The input language of each machine is a set of conditioned events. A condition is a predicate on monitored or mode class variables. For example, let condition SwitchOn represent predicate [On/Off switch = On], and condition PumpFail represent predicate [Pump failure = true]. Representing conditions as predicates allows us to assume that all conditions are boolean variables.

A machine's behavior is defined and controlled by changes to the values of conditions, called events. We assume that conditions can change value at most once during a single time unit. Events are detectable only at the point in time in which they occur. For example, events @T(a) and @F(a) represent a condition a becoming true and becoming false, respectively. a is the event's triggering condition. All events in the form @T(a) or @F(a) are called primitive events.

Compound events can be specified using boolean operators, connecting primitive events or multiple conditions. The following two formulas represent different events:

$$@\mathrm{T}(\mathrm{A} \wedge \mathrm{B})$$

$$@\mathrm{T}(\mathrm{A}) \wedge @\mathrm{T}(\mathrm{B})$$

The first formula describes an event occurring at the point in time when the conjunction of the conditions becomes true. This means that immediately before this time, A $\wedge$ B was false (i.e., at most one of A and B was true). The second formula describes two simultaneous events occurring at the point of time when both conditions become true. Immediately before the event, neither condition is true. Some formal semantics, e.g.

[HLK95], exclude simultaneous events. We assume that simultaneous events can only occur between variables related by environmental assumptions. (See Section 2.3 for description of environmental assumptions and Section 3.2.5 for algorithms to enforce these assumptions.)

A conditioned event

$$@\mathrm{T}(a) \text{ WHEN } [b]$$

describes the point of time when a becomes true while b remains true. Since time is discrete, we can reason about the current and the previous state. If "current" is the state of the system at time t, then "previous" is the state at time $t - \epsilon$ for some well-defined ϵ - a unit of measurement of time. The logical statement represented by a conditioned event is defined by[HBGL95]:

$$@\mathrm{T}(a) \text{ WHEN } [b] = \neg\, a \wedge a\prime \wedge b,$$

where the unprimed version of a condition a denotes the value of a in the previous state and the primed version denotes the value of a in the current state. b is called the event's WHEN condition.

2.2.2 Modes and mode transitions

A state of the monitored environment is a mapping of variables to values, and a state space is the set of possible combinations of values of variables. However, the behavior of the system is rarely affected by the values of all the variables at once. A mode class defines a set of states, called modes, that partition the monitored environment's state space. An SCR requirements document contains the specification of one or more mode classes. At all times, the system is in exactly one mode of each mode class.

Each mode class specifies one aspect of the system's behavior, and the system's global behavior is defined to be the composition of the specification's mode classes. Transitions between pairs of modes are activated by conditioned events. If a conditioned event can trigger two or more transitions from the same mode, then the mode class is non-deterministic. One mode of each modeclass is designated as the initial mode. Assumptions about the initial state of the environment are specified with the initial mode. We can think of a mode class as an enumerated type whose values are modes of that mode class.

Current Mode	r_1	r_2	...	r_n	**New Mode**
m_1	$e_{1,1,1}$	$e_{1,2,1}$	...	$e_{1,n,1}$	$m_{1,1}$
	$e_{1,1,2}$	$e_{1,2,2}$	...	$e_{1,n,2}$	$m_{1,2}$
	...	...	...	...	...
	$e_{1,1,k_1}$	$e_{1,2,k_1}$	...	e_{1,n,k_1}	m_{1,k_1}
...	...	...	...	...	...
m_l	$e_{l,1,1}$	$e_{l,2,1}$	...	$e_{l,n,1}$	$m_{l,1}$
	$e_{l,1,2}$	$e_{l,2,2}$	...	$e_{l,n,2}$	$m_{l,2}$
	...	...	...	...	...
	$e_{l,1,k_l}$	$e_{l,2,k_l}$	...	e_{l,n,k_l}	m_{1,k_l}

Initial: $m_i(cond)$

Table 2.1: General format of a mode transition table.

The specification of a mode class's transition relation has a tabular format, and is called a mode transition table. In Table 2.1, each row specifies a conditioned event that activates a transition from the mode on the left to the mode on the right, i.e., a transition between mode m_i and mode $m_{i,k}$ occurs as the result of the combination of events $e_{i,1,k}$ .. $e_{i,n,k}$ Each table entry $e_{i,j,k}$ is one of "@T", "@F", "t", "f", or "–".

- A value "@T" represents the triggering condition for the transition, indicating that a monitored variable r_j, labeling the column, is changing value from false to true. "@F" indicates a change of r_j's value from true to false.

- A value "t" (or "f") represents a WHEN condition for the transition, meaning that the variable r_j had the value true (false) when the transition originated.

- A hyphen ("–") indicates that this condition does not affect the occurrence of the conditioned event.

Below the mode transition table is the specification of the system's initial mode. The system starts in mode m_i if *cond*, some condition on values of monitored variables, holds. Otherwise, the initial state of the system is undefined.

Mode	Triggering Event	
m_1	$e_{1,1}$	$e_{1,2}$
m_2	$e_{2,1}$	$e_{2,2}$
...	...	...
m_l	$e_{l,1}$	$e_{l,2}$
$r_j =$	true	false

Initial: $v(cond)$

Table 2.2: General format of an event table.

Values of controlled variables change in response to events when the system is in particular modes. These changes are captured in <u>event tables</u>, one for each controlled variable. Since we assume that each controlled variable is boolean, our event tables consist of 3 columns. Table 2.2 shows the general format of an event table for a variable r_j. A conditioned event is triggered by a change in a monitored variable when the system is in a particular mode (shown in the left-most column). The second (third) column is used to specify conditions changing r_j's value to true (false). For example, when the system is in mode m_2 and an event $e_{2,1}$ occurs, the value of r_j is true. Each table entry $e_{i,j}$ represents a conditioned event, so a typical entry is

$$@\mathrm{T}(a) \text{ WHEN}[b].$$

To indicate that the transition can be triggered when the system enters a particular mode, we use the InMode() function [van90]. For example, if $e_{1,1}$ were an event @T(InMode), then the change in r_j's values is triggered by the system entering mode m_1.

Below the event table is a starting value for the controlled variable. The starting value of r_j is v (v is true or false) if *cond* holds. Otherwise, it is undefined.

Definition 2.2 *Behavioral requirements $\mathcal{B}$ expressed in SCR is a tuple $\mathcal{B} = \langle R, V, T, MT, CV, L \rangle$, where*

$R = \langle M, C, MD \rangle$ is a set of system variables. M is a set of monitored, C is a set of controlled, and MD is a set of modeclass variables;

V is a set of values that variables in R can attain;

T: $R \rightarrow 2^V$ is a mapping of variables to values:

$\forall\ r_j \in M \cup C,\ T(r_j) = \{true, false\}$

$\forall\ md_j \in MD,\ T(md_j)=\{m_{j,1},\ m_{j,2}, \ldots ,\ m_{j,n}\}$, where $m_{j,k}$ is a mode in the modeclass md_j.

L is the set of events;

MT: $MD \times L \rightarrow MD$ is a relation describing mode transitions; and

CV: $C \times MD \times L \rightarrow C$ is a relation describing changes of values of controlled variables.

$|R|$ indicates the number of variables in the system, and $\forall r \in R$, $|T(r)|$ indicates the number of values r can attain. We assume that variables are boolean and finite in number, and that each mode class contains a finite number of modes. Thus, the state space S described by these requirements is finite and bounded above by

$$S = 2^{|M \cup C|} \times \Pi_{1 \leq j \leq |MD|} |T(md_j)|$$

2.3 Environmental Assumptions

An assumption, the NAT relation in the Four Variable Model, specifies constraints on the values of variables, imposed either by laws of nature or by other mode classes in the system. As such, assumptions are invariant constraints that must hold in all system states. For example, the water in a container cannot be too high and too low at the same time, and buttons can be either pressed or released but not both.

[Atl92] identified several types of assumptions on environmental conditions and described their syntax and semantics. The environmental assumptions were designed to help detail SCR tables, so their semantics is presented in the form of an implication, i.e., what can be derived about one condition if we know that another condition is true, false, becoming true, or becoming false.

implication ($a\text{-}{>}\ b$). This assumption indicates that the state space in which a is true is a subset of the state space in which b is true, resulting in the following six relations:

$$\begin{gathered}
a \rightarrow b \\
@T(a) \rightarrow (b \ \vee \ @T(b)) \\
@F(a) \rightarrow (b \ \vee \ @F(b)) \\
\neg b \rightarrow \neg a \\
@T(b) \rightarrow (\neg a \ \vee \ @T(a)) \\
@F(b) \rightarrow (\neg a \ \vee \ @F(a))
\end{gathered}$$

equivalence ($a = b$). This assumption indicates that $a\!-\!\!> b$ and $b\!-\!\!> a$, i.e., the state space in which a is true is equal to the state space in which b is true, resulting in the following eight relations:

$$
\begin{gathered}
a \rightarrow b \\
\neg a \rightarrow \neg b \\
@T(a) \rightarrow @T(b) \\
@F(a) \rightarrow @F(b) \\
\neg b \rightarrow \neg a \\
b \rightarrow a \\
@T(b) \rightarrow @T(a) \\
@F(b) \rightarrow @F(a)
\end{gathered}
$$

strict implication ($a\!-\!\!>> b$). This assumption is similar to implication except that when a becomes true, b should already be true, and when b becomes false, a should already be false. The assumption results in the following six relations:

$$
\begin{gathered}
a \rightarrow b \\
@T(a) \rightarrow b \\
@F(a) \rightarrow b \\
\neg b \rightarrow \neg a \\
@T(b) \rightarrow \neg a \\
@F(b) \rightarrow \neg a
\end{gathered}
$$

timeline(a < b). Conditions a and b represent lengths of time that a particular environmental condition has been true, where b represents a longer length of time than a. The timeline assumption is a strict implication assumption whose contrapositive is non-strict implication. Therefore, a must already be true when b becomes true and must become false when b becomes false. This assumption results in the following six relations:

$$b \rightarrow a$$
$$@T(b) \rightarrow a$$
$$@F(b) \rightarrow @F(a)$$
$$\neg a \rightarrow \neg b$$
$$@T(a) \rightarrow \neg b$$
$$@F(a) \rightarrow (\neg b \ \vee \ @F(b))$$

enumeration $(a_0 \mid a_1 \mid \cdots \mid a_n)$. The state space is partitioned such that exactly one member of the enumeration is true in each partition. Some of the relations that an enumeration $a_0 \mid a_1 \mid a_2$ results in are shown below:

$$a_1 \rightarrow (\neg a_2 \ \wedge \ \neg a_3)$$
$$a_2 \rightarrow (\neg a_1 \ \wedge \ \neg a_3)$$
$$a_3 \rightarrow (\neg a_1 \ \wedge \ \neg a_2)$$

For an algorithm to enforce these relations, please refer to Section 3.2.5. [Atl92] also identified a number of other relations.

The assumptions about environmental conditions can be found in the section of an SCR requirements document that describes the system's controlled and monitored variables. A human requirements analyzer is responsible for accurately interpreting the environmental assumptions as they are stated in the specification and translating them into the appropriate syntax.

2.4 System Goals

In addition to operational specification, SCR requirements usually contain a set of <u>goals</u> that the system must meet. These goals are not additional constraints on the required behavior; rather, they are redundant information included in the specification because the reader might not deduce these properties from the tabular specifications. These goals may be mode invariants or reachability information, express global behavioral requirements on the occurrence of an event, or represent temporal properties. [Atl92] introduces a simple language for capturing some of these global properties compactly. In the following list, P, P_1 and P_2 represent propositional logic formulae:

- $smi(Mc = Md, P)$ (strong mode invariant) is satisfied if P holds in every state where the system is in mode Md of modeclass Mc.

- $wmi(Mc = Md, P)$ (weak mode invariant) is satisfied if P holds when the system is in mode Md of modeclass Mc, but not while the system is exiting mode Md.

- $cause(P_1, P_2)$ is satisfied if whenever P_1 is True, either P_2 is True or the next transition of the system will establish a state in which P_2 holds.

- $strcause(P_1, P_2)$ is satisfied if whenever P_1 is True, the next transition of the system will establish a state in which P_2 holds.

- $reach(P)$ is satisfied if the system can reach a state in which P holds. reach properties are usually used to ensure that other types of properties are not satisfied vacuously.

These types of user-specified properties frequently appear in requirements documents, and during our case studies we did not need to verify any other types of properties. Requirements specifications of realistic size and complexity (like [AFB+88]), however, contain global properties which cannot be expressed using only these assertions. We defer a further discussion of this issue until Chapter 4.

2.5 Example - SWLMS

Consider a simple Water-Level Monitoring System (SWLMS). A switch controls whether the system is on or off. If the system is on and its sensors detect too much (too little) water, a pump is turned on for a fixed period to remove (add) some water. If the sensor or the pump fails, the system enters an error state. This simplified version of the system has no error recovery, so there are no transitions from the error state. Such a system could be represented by a single mode class, called MC, comprised of three modes:

Off	The system is Off.
Operating	The system is working properly.
Error	The system has failed.

We need four monitored variables to determine if the water level is too low or too high, to detect if the system is turned on, and to detect if the pump failed. We also need a controlled variable, PumpOn, to activate the pump. The variables are described below:

SwitchOn	Is the switch on?
PumpFail	Did the pump fail?
TooHigh	Is there too much water?
TooLow	Is there too little water?
PumpOn	Should the system turn the pump on?

Table 2.3 shows a mode transition table for SWLMS. Mode class MC starts in mode Off, and all monitored variables are initially false. Assume the system is Off and no pump failure has been detected. When the user turns the switch on, the system goes

to mode Operating. As soon as the pump failure is detected, the system goes to mode Error. Turning the switch off while the system is in Operating causes a transition to mode Off.

Current Mode	SwitchOn	PumpFail	**New Mode**
Off	@T	f	Operating
	–	@T	Error
Operating	@F	f	Off
	–	@T	Error

Initial: Off ($\neg$SwitchOn $\wedge$ $\neg$PumpFail $\wedge$ $\neg$TooHigh & $\neg$TooLow)

Assumptions: TooLow–>> $\neg$TooHigh

Table 2.3: Mode transition table for modeclass MC of SWLMS.

Table 2.4 shows an event table for the controlled variable PumpOn. This variable starts with value false and becomes true when the system is in mode Operating and either event @T(TooHigh) or @T(TooLow) occurs. It becomes false when the water is within limits or when pump fails.

Mode	Triggering Event	
Operating	@T(TooHigh)	–
	@T(TooLow)	–
	–	@F(TooHigh)
	–	@F(TooLow)
	–	@T(PumpFail)
Off	–	@T(PumpFail)
PumpOn =	true	false

Initial: False

Table 2.4: Event table for controlled variable PumpOn.

The assumption specified in Table 2.3:

$$\text{TooLow} \rightarrow\!\!> \neg\text{TooHigh}$$

states that the water level cannot be too high and too low at the same time, i.e. if TooHigh is true, TooLow is false, and vice versa.

Most of the goals specified in the SWLMS example are mode invariants:

- If the system is in mode Off, then variables SwitchOn and PumpFail are false.

Some other goals express global behavioral requirements when an event occurs:

- If event @T(SwitchOn) occurs, the system cannot remain in mode Off.

while others specify temporal properties:

- When the water level becomes too high while the system is in mode Operating, the pump will immediately be turned on.

The following is the list of system goals of SWLMS:

- reach(MC=Operating) indicates that the system can eventually transit to mode Operating.

- wmi(MC=Operating, SwitchOn). This property holds even though an event @F(SwitchOn) occurs when the system is in mode Operating since the system leaves this mode in all of its next states.

- cause((MC=Operating) $\wedge$ TooHigh, PumpOn) indicates that as soon as the system reaches a state where it is in Operating and the water level is too high, the pump is immediately turned on in all next states.

The full specification of SWLMS can be found in Appendix A.1.

2.6 Discussion

Our assumptions that monitored and controlled variables are boolean simplifies the format of mode transition and event tables. This assumption is not always realistic. Environmental assumptions allow us to express first-order predicates (by a finite number of related boolean conditions), like integer ranges. However, we are not able to process a richer type system.

The model presented above is somewhat different from the one recently proposed by Naval Research Lab[HLK95, HBGL95]. Their model has a rich type system, so there is no need for environmental assumptions. They also make a "One Input Assumption", i.e., only one monitored variable can change value at one time, so no simultaneous events occur. This assumption allows for a clean formalization and easy simulation of requirements.

2.7 Consistency with SCR Requirements

Definition 2.3 *Let a set of SCR requirements $\mathcal{R} = \langle \mathcal{B}, \mathcal{E}, \mathcal{G} \rangle$ be given. A program artifact $\mathcal{A}$ constrained by environmental assumptions $\mathcal{E}$ (called $\mathcal{A}_\mathcal{E}$) is consistent with its requirements $\mathcal{R}$ if*

1. *$\mathcal{A}_\mathcal{E}$ implements all state transitions specified in the $\mathcal{B}$, and*
2. *$\mathcal{A}_\mathcal{E}$ does not implement any state transitions which are not specified in $\mathcal{B}$, and*
3. *$\mathcal{A}_\mathcal{E}$ satisfies all user-defined safety properties $\mathcal{G}$.*

This is a very restricted definition. Typically, artifacts are considered consistent with requirements when they implement at least what is specified. SCR was developed to specify high-assurance systems, and intended to capture all allowed system behaviors. Indeed, it is clearly a fault if an artifact implements an unspecified transition to a state representing a failure.

A requirements specification can be translated into a list of properties which capture this notion of consistency. To prove that an artifact is consistent with its requirements, we demonstrate that it is a model of all of all these properties. We express these properties as first-order logic formulas.

2.7.1 Formal model of SCR

Before expressing the consistency properties in a formal logic, we need to define the meaning of a mode transition and a change of value of a controlled variable. [HLK95] describes a mode change as a transition between two states, i.e. a transition between modes m_i and m_j of modeclass m triggered by @T(a) WHEN [b] is formalized as

$$(m = m_i) \ \wedge \ (a = \text{false}) \ \wedge \ (b = \text{true}) \ \wedge \ (a\prime = \text{true}) \ \wedge \ (m\prime = m_j),$$

where a condition represents its value in the previous state, and a primed condition represents its value in the current state. In this formalization, the mode transition happens concurrently with the event being triggered.

We, on the other hand, are interested in defining a semantics that will enable verification that detailed designs of programs correspond to SCR requirements, and our semantics captures the way people write code: first a monitored variable is assigned a value; then tests determine what value was assigned and whether an event occurred. If so, the system's mode changes. We use states to denote points at which variables change values. Thus, three states need to be considered: the state in which the monitored variable had its original value, the state in which the monitored variable was assigned a new value, and the state in which the mode change occurred. Generally, we do not know the value assigned to a monitored variable until after the outcome of a test. Thus, monitored variables may have either the value true or the value false in a state; the exact value not being known until an arc representing a successful test outcome is traversed. As the result, we define formulas as being true on arcs as well as in states. A transition between m_i and m_j triggered by @T(a) WHEN [b] is formalized as

$$(a = \text{false}) \ \wedge \ (m\prime = m_i) \ \wedge \ (b\prime = \text{true}) \ \wedge \ (a\prime = \text{true}) \ \wedge \ (m\prime\prime = m_j),$$

where a condition represents its value on the previous edge, a primed condition represents its value on the current edge, and the double-primed condition represents its value in the adjacent state. Figure 2.2 gives a pictorial representation of this semantics.

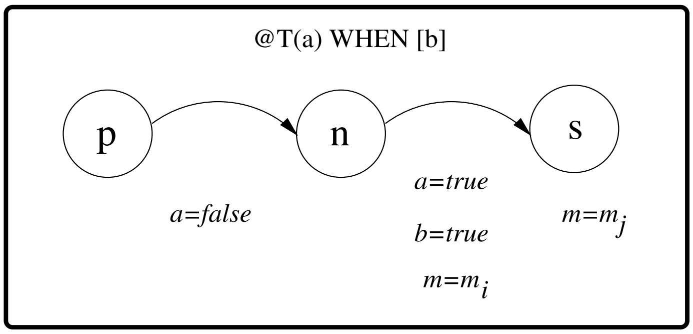


Figure 2.2: Pictorial representation for a mode transition.

For a state n and a formula f, we use the notation $n \models f$ to indicate that f is true in n. For a pair of states n, p, we use the notation $(n, p) \models f$ to indicate that f is true on an edge between n and p. We also assume that for each state n we have functions pred(n) and succ(n) returning a list of all predecessors and successors of n, respectively. (This list can be empty.) Thus, we can express a property "there exists a transition from $m = m_i$ to $m = m_j$ on @T(a) WHEN [b]" as

$$\exists n,\ \exists s \in \text{succ}(n),\ \exists p \in \text{pred}(n),\ (s \models (m = m_j)) \wedge$$
$$((n, s) \models ((a = \text{true})\ \wedge\ (b = \text{true})\ \wedge\ (m = m_i)))\ \wedge\ ((p, n) \models (a = \text{false}))$$

A number of properties generated from SCR have the notion of an event in them. We say that an event @T(a), where a is a boolean variable, has occurred on an edge (n, s), i.e. $(n, s) \models$ @T(a), if

$$((n, s) \models (a = \text{true}))\ \wedge\ (\exists p \in \text{pred}(n),\ (p, n) \models (a = \text{false}))$$

We say $(n, s) \models$ @F(a), if

$$((n, s) \models (a = \text{false}))\ \wedge\ (\exists p \in \text{pred}(n),\ (p, n) \models (a = \text{true}))$$

For a modeclass mc, we say that an event @T($mc = md$) has occurred on an edge (n, s), if

$$((n, s) \models (mc = md))\ \wedge\ (\exists p \in \text{pred}(n),\ (p, n) \models (\neg(mc = md)))$$

We say $(n, s) \models$ @F($mc = md$), if

$$((n, s) \models (\neg(mc = md)))\ \wedge\ (\exists p \in \text{pred}(n),\ (p, n) \models (mc = md))$$

2.7.2 Automatically-Generated Safety Properties

A set of properties capturing first two parts of our definition of consistency can be obtained by composing rows and columns of SCR tables. In Section 2.5 we introduced a simple Water-Level Monitoring System (SWLMS). We use the SWLMS requirements to illustrate the kinds of properties which are generated to capture our notion of consistency with SCR requirements. For example, we have a property asserting that the only way for mode MC to be in mode Off in its next state is if MC is currently in Off, or if a transition from mode Operating occurs in response to an event @F(SwitchOn) WHEN ¬PumpFail. This property was obtained by composing the rows of the MC mode transition table which have Off in their right columns (in this case, only row three). We write this property as

$$\begin{aligned} P_1 \quad = \quad & \forall n,\ n \models (\mathrm{MC{=}Off}) \rightarrow (\forall p \in \mathrm{pred}(n),\ (p, n) \models (\mathrm{MC{=}Off}) \\ & \vee\ (p, n) \models (@\mathrm{F(SwitchOn)} \wedge (\mathrm{PumpFail{=}false}) \wedge (\mathrm{MC{=}Operating}))) \end{aligned}$$

Here we are using @F(SwitchOn) as the shorthand notation to express events.

Properties quantified on all members pred(n) or succ(n) are considered vacuously true when the corresponding list is empty.

We generate properties similar to P_1 for each value of controlled variables and every mode in the right columns of mode transition tables. These properties capture the second part of our notion of consistency and are called "only legal transitions" (OLT) properties. OLT properties generated from requirements of SWLMS are shown in Figure 2.3. Property P_2 asserts that if the system will be in mode Operating in its next state, then it either must be in this mode in the current state, or the transition from Off on @F(SwitchOn) WHEN [PumpFail=false] is occurring. P_2 was generated from row one of Table 2.3, and P_3 from a composition of rows two and four. Properties P_4 and P_5 were generated for the controlled variable PumpOn from Table 2.4. There are two OLT properties generated for each controlled variable, reflecting changes of value to false (P_4) and to true (P_5).

Another property asserts that there exists a transition from mode Off to mode Operating on an event @T(SwitchOn) WHEN [PumpFail=false]. This property corresponds

$$
\begin{aligned}
P_1 &= \forall n,\ n \models (\text{MC=Off}) \rightarrow (\forall p \in \text{pred}(n),\ (p,n) \models (\text{MC=Off}) \\
&\qquad \vee\ (p,n) \models (\text{@F(SwitchOn)} \wedge (\text{PumpFail=false}) \wedge (\text{MC=Operating}))) \\
P_2 &= \forall n,\ (n \models (\text{MC=Operating}) \rightarrow (\forall p \in \text{pred}(n),\ (p,n) \models (\text{MC=Operating}) \\
&\qquad \vee\ (p,n) \models (\text{@T(SwitchOn)} \wedge (\text{PumpFail=false}) \wedge (\text{MC=Off}))) \\
P_3 &= \forall n,\ (n \models (\text{MC=Error}) \rightarrow (\forall p \in \text{pred}(n),\ (p,n) \models (\text{MC=Error})\ \vee \\
&\quad (p,n) \models (\text{@T(PumpFail)} \wedge (\text{MC=Operating}))\ \vee \\
&\quad (p,n) \models (\text{@T(PumpFail)} \wedge (\text{MC=Off}))) \\
P_4 &= \forall n,\ (n \models (\text{PumpOn=false}) \rightarrow (\forall p \in \text{pred}(n),\ (p,n) \models (\text{PumpOn=false})\ \vee \\
&\quad (p,n) \models (\text{@F(TooHigh)} \wedge (\text{MC=Operating}))\ \vee \\
&\quad (p,n) \models (\text{@F(TooLow)} \wedge (\text{MC=Operating}))\ \vee \\
&\quad (p,n) \models (\text{@T(PumpFail)} \wedge (\text{MC=Operating}))\ \vee \\
&\quad (p,n) \models (\text{@T(PumpFail)} \wedge (\text{MC=Off}))) \\
P_5 &= \forall n,\ (n \models (\text{PumpOn=true}) \rightarrow (\forall p \in \text{pred}(n),\ (p,n) \models (\text{PumpOn=true})\ \vee \\
&\quad (p,n) \models (\text{@T(TooHigh)} \wedge (\text{MC=Operating}))\ \vee \\
&\quad (p,n) \models (\text{@T(TooLow)} \wedge \text{MC=Operating}))
\end{aligned}
$$

Figure 2.3: OLT properties for SWLMS.

to the first row of Table 2.3. We express this property as

$$
\begin{aligned}
P_6 &= \exists n,\ \exists p \in \text{pred}(n),\ n \models (\text{MC=Operating}) \wedge \\
&\qquad (p,n) \models ((\text{MC=Off}) \wedge \text{@T(SwitchOn)} \wedge (\text{PumpFail=false}))
\end{aligned}
$$

Such properties ensure that all transitions specified in the requirements appear in the design, capturing the first part of our notion of consistency. We call them "all legal

$$
\begin{aligned}
P_6 &= \exists n,\ \exists p \in \mathrm{pred}(n),\ n \models (\mathrm{MC{=}Operating}) \wedge \\
&\quad (p,n) \models ((\mathrm{MC{=}Off}) \wedge @\mathrm{T}(\mathrm{SwitchOn}) \wedge (\mathrm{PumpFail{=}false})) \\
P_7 &= \exists n,\ \exists p \in \mathrm{pred}(n),\ n \models (\mathrm{MC{=}Error}) \ \wedge\ (p,n) \models ((\mathrm{MC{=}Off}) \\
&\quad \wedge @\mathrm{T}(\mathrm{PumpFail})) \\
P_8 &= \exists n,\ \exists p \in \mathrm{pred}(n),\ n \models (\mathrm{MC{=}Off}) \wedge \\
&\quad (p,n) \models ((\mathrm{MC{=}Operating}) \wedge @\mathrm{F}(\mathrm{SwitchOn}) \wedge (\mathrm{PumpFail{=}false})) \\
P_9 &= \exists n,\ \exists p \in \mathrm{pred}(n),\ n \models (\mathrm{MC{=}Error}) \wedge \\
&\quad (p,n) \models ((\mathrm{MC{=}Operating}) \wedge @\mathrm{T}(\mathrm{PumpFail})) \\
P_{10} &= \exists n,\ \exists p \in \mathrm{pred}(n),\ n \models (\mathrm{PumpOn{=}false}) \wedge \\
&\quad (p,n) \models ((\mathrm{PumpOn{=}true}) \wedge @\mathrm{F}(\mathrm{TooHigh}) \wedge (\mathrm{MC{=}Operating})) \\
P_{11} &= \exists n,\ \exists p \in \mathrm{pred}(n),\ n \models (\mathrm{PumpOn{=}false}) \wedge \\
&\quad (p,n) \models ((\mathrm{PumpOn{=}true}) \wedge @\mathrm{F}(\mathrm{TooLow}) \wedge (\mathrm{MC{=}Operating})) \\
P_{12} &= \exists n,\ \exists p \in \mathrm{pred}(n),\ n \models (\mathrm{PumpOn{=}false}) \wedge \\
&\quad (p,n) \models ((\mathrm{PumpOn{=}true}) \wedge @\mathrm{T}(\mathrm{PumpFail}) \wedge (\mathrm{MC{=}Operating})) \\
P_{13} &= \exists n,\ \exists p \in \mathrm{pred}(n),\ n \models (\mathrm{PumpOn{=}false}) \wedge \\
&\quad (p,n) \models ((\mathrm{PumpOn{=}true}) \wedge @\mathrm{T}(\mathrm{PumpFail}) \wedge (\mathrm{MC{=}Off})) \\
P_{14} &= \exists n,\ \exists p \in \mathrm{pred}(n),\ n \models (\mathrm{PumpOn{=}true}) \wedge \\
&\quad (p,n) \models ((\mathrm{PumpOn{=}false}) \wedge @\mathrm{T}(\mathrm{TooHigh}) \wedge (\mathrm{MC{=}Operating})) \\
P_{15} &= \exists n,\ \exists p \in \mathrm{pred}(n),\ n \models (\mathrm{PumpOn{=}true}) \wedge \\
&\quad (p,n) \models ((\mathrm{PumpOn{=}false}) \wedge @\mathrm{T}(\mathrm{TooLow}) \wedge (\mathrm{MC{=}Operating}))
\end{aligned}
$$

Figure 2.4: ALT properties for SWLMS.

transitions"(ALT) properties. One ALT property is generated for every row of transition tables for modeclasses and controlled variables. Other ALT properties for SWLMS are shown in Table 2.4. Properties P_6-P_9 were generated from Table 2.3 (rows 1-4, respectively). Properties P_{10}-P_{15} were generated from Table 2.4 (rows 1-6, respectively).

2.7.3 User-Specified Safety Properties

The third part of the notion of consistency is represented by global properties. Since these properties are specified by users rather than generated from tables, we call them "user-specified properties" (USER):

$$
\begin{aligned}
U_1 = \text{smi}(\text{Mc=Md, P}) &= \forall n,\ (n \models (\text{Mc=Md})) \rightarrow (n \models \text{P}) \\
U_2 = \text{wmi}(\text{Mc=Md, P}) &= \forall n,\ \forall p \in \text{succ}(n) \\
&\quad\ ((n \models (\text{Mc=Md}))\ \wedge (p \models (\text{Mc=Md}))) \rightarrow n \models \text{P} \\
U_3 = \text{cause}(\text{P}_1, \text{P}_2) &= \forall n,\ (n \models \text{P}_1) \rightarrow (n \models \text{P}_2\ \vee\ (\forall p \in \text{succ}(n),\ p \models \text{P}_2)) \\
U_4 = \text{strcause}(\text{P}_1, \text{P}_2) &= \forall n,\ n \models \text{P}_1 \rightarrow (\forall p \in \text{succ}(n),\ p \models \text{P}_2) \\
U_5 = \text{reach}(\text{P}) &= \exists n,\ n \models P, \text{ where } n \text{ is reachable from starting state}
\end{aligned}
$$

This translation clearly identifies the difference between *smi* and *wmi* properties. Property smi(Mc=Md, P) means that for every state of the system P holds whenever the system is in mode Md. In contrast, property wmi(Mc=Md, P) means that P should hold in cases where the system is in mode Md in the current and in all the immediate successor states. User-specified properties for SWLMS system are shown in Figure 2.5. Let $\mathcal{P}$ be the set of all properties generated from an SCR specification. For the SWLMS example, $\mathcal{P} = \{P_1, ..., P_{18}\}$.

$$
\begin{aligned}
P_{16} &= \text{reach}(\text{MC=Operating}) = \exists n,\ n \models (\text{MC=Operating}) \\
P_{17} &= \text{wmi}(\text{MC=Operating, SwitchOn}) \\
&= \forall n,\ (n \models (\text{MC=Operating})\ \wedge (\forall p \in \text{succ}(n),\ p \models (\text{MC=Operating}))) \\
&\quad \rightarrow n \models \text{PumpOn} \\
P_{18} &= \text{cause}((\text{MC=Operating}) \wedge \text{TooHigh, PumpOn}) \\
&= \forall n,\ (n \models (\text{MC=Operating})\ \wedge n \models \text{TooHigh}) \rightarrow (n \models \text{PumpOn} \vee \\
&\quad \forall p \in \text{succ}(n),\ p \models \text{PumpOn})
\end{aligned}
$$

Figure 2.5: User-specified properties for SWLMS.

Chapter 3

Creating an Abstraction

In this chapter we describe the program design language for creating detailed designs and a process for building a finite-state abstraction from the design. The detailed design is used to create a Design Flow Graph (DFG) which is abstracted into a Finite-State Machine (FSM). Afterwards, the FSM is used to check system properties. Figure 3.1 gives a roadmap of the analysis process described in this and the next chapters.

3.1 Design

A Program Design Language (PDL)[CG75] is a language used to specify designs. We have defined our own PDL to reason about designs for SCR requirements. Our PDL was motivated by a number of factors:

- Designs should look like real programs, i.e. specify control flow via programming language constructs for sequence, selection, and iteration.
- Designs should capture the <u>essence</u> of what is happening in the code, rather than details[Bro87]. When writing designs, we want to reason about requirements-level variables rather than implementation-level structures.
- Finally, designs should be able to deal with sensors and actuators of the system, i.e., with IN and OUT relations (see Chapter 2 and [PM95]).

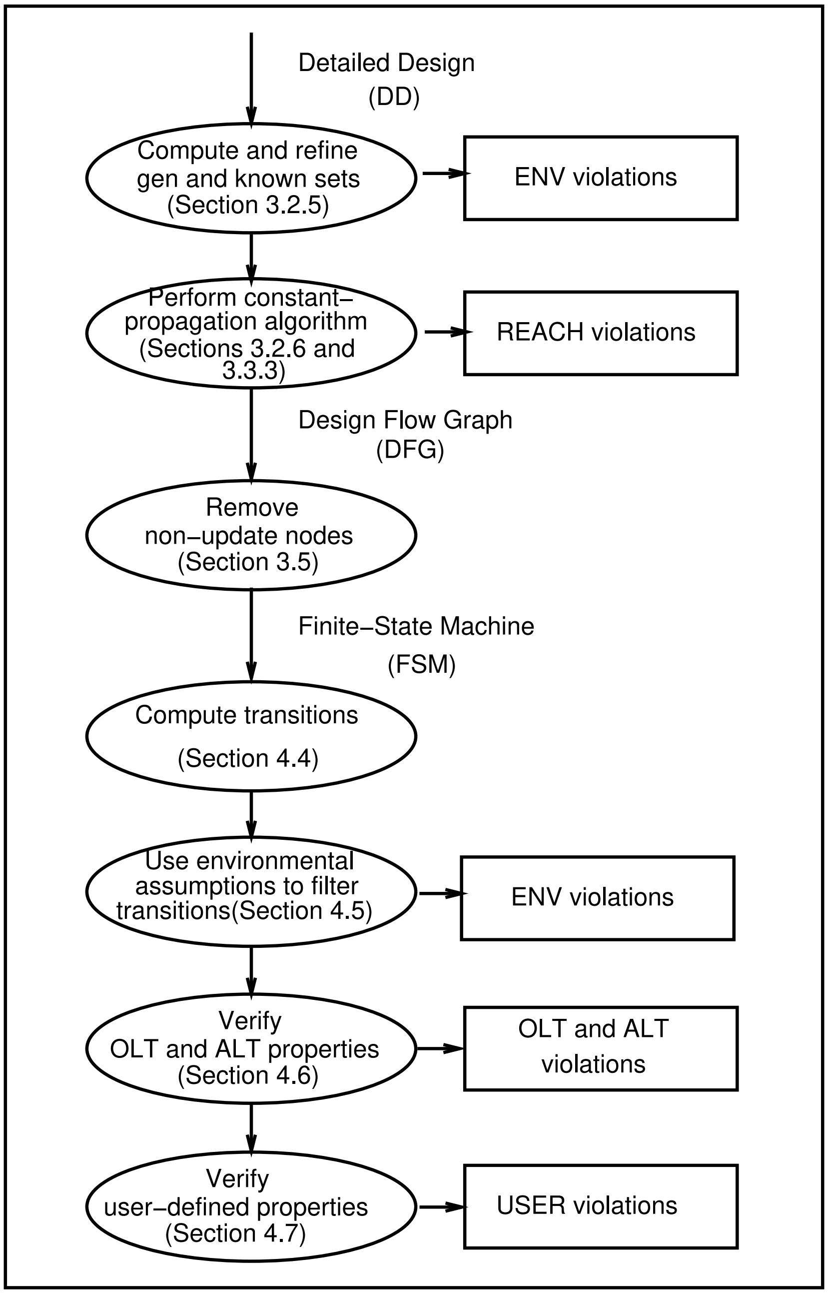


Figure 3.1: Analysis roadmap.

3.1.1 Design Constructs

Typically, control-flow-based PDLs[CG75] are defined by an <u>outer syntax</u> of control structures and <u>inner syntax</u> of other statements. Our PDL's outer syntax is a set of

C-like control structures. Our inner syntax consists of annotations - special statements describing values of requirements variables. Our use of annotations was inspired by William Howden's work on QDA[HW94, Vai91].

For sequential designs, we defined three types of annotations:

- An Initial annotation indicates the starting state of each mode class. It unconditionally assigns values to variables. This annotation corresponds to initialization information specified in the requirements.

- An Update annotation assigns values to variables, identifying points at which the program changes its state.

- An Assert annotation reflects a programmer's knowledge that variables have particular values in the current state. Static analysis usually gives imprecise results because states are aggregated. Assert annotations reduce the amount of information in the state to what the programmer believes to be true.

The syntax of Update annotations is described below:

Annotation Syntax	Alternative Syntax	Meaning
Update A=true	Update A	A receives the value {true}.
Update A=false	Update ~A	A receives the value {false}.
Update A=Top	Update Top(A)	A receives the value corresponding to the top of the lattice ({true,false} if A is binary).
Update M=M1	Update M(M1)	Modeclass M receives the value {M1}.

Either syntax can be used when writing designs. Variables in Update annotations may be combined using the & (AND) operator, indicating that all variables receive their values at the same time. The syntax of Assert annotations is the same as that of Update annotations, except that variables may also be combined using the | (OR) operator, indicating that the programmer knows (or assumes) that at least one of the disjuncts is true.

A designer might also use other statements in addition to annotations and control-flow constructs. We do not process the former since they do (should) not reflect changes of values of requirements variables. To differentiate between statements and annotations, the latter start with @@. For processing concurrent designs, we extended the PDL to contain synchronization and communication information. We discuss these changes in Chapter 6. The complete syntax of our PDL is shown in Appendix B.

The following design fragment shows sample uses of annotations:

```
ReadDeviceMonitor();     /* read value */
@@ update PumpFail=Top;
if (pump_failed()) {   /* test value */
    @@ Assert PumpFail=true;
    @@ Update MC=Error;
}
else {
   @@ Assert PumpFail=false;
   ...
}
```

In this fragment, the function ReadDeviceMonitor() is called to determine the status of a pump. The Update annotation records the outcome of this call by setting the value of the requirement's variable PumpFail to Top (see Section 3.2.2). The function pump_failed() determines if the value read corresponds to failure of the pump. We assert that the Then clause will be executed only if the pump did fail (i.e., the value of PumpFail is true rather than Top). In this case, the system should transit to mode Error. In the Else branch, we assert that the value of PumpFail is false rather than

Top.

3.1.2 How to write designs

Designs of programs implementing SCR requirements are very stylized: an Initial annotation marks the starting state and is followed by a loop in which the system reads relevant monitors, tests their values and then decides to change either its mode or the values of several controlled variables. Figure 3.2 shows the design of the Simplified Water-Level System from Chapter 2 written in this style. Assert annotations are used to document the results of testing predicates, and Update annotations mark changes in the program's state. In Figure 3.2, controlled variables receive values on lines 17, 34, 38 and 44. Such a design could have been automatically generated. Unfortunately, generating designs for complex systems with several mode classes is not a trivial task, and we make no attempt to automate it. Experienced designers might want to hand-optimize their designs by "factoring out" common actions for different states. Figure 3.3 shows the result of such an optimization. Here we notice that the pump can be turned on only when the system is in mode Operating. So, the design has an inner while loop, in which the system reads the water level and determines the state of the pump. The system exits the loop when the pump fails or when the switch is turned Off. Since the SWLMS has no error recovery, transitions to mode Error are done outside the main while loop.

After a detailed design is complete and verified, it can be further refined into an implementation. During our experiments with designing and implementing various systems, we discovered that in most cases the code and the design are closely related.

```
{ ...
  @@ Initial MC=Off & SwitchOn=false & PumpFail=false &
     TooHigh=false & TooLow=false & PumpOn=false;
  while (1) {
    if (IN_OFF(System)) {
      @@ Assert MC=Off;  /* the system is in mode Off */
      READ_SWITCH();     /* read switch monitor       */
      @@ Update SwitchOn=Top;
      if (SWITCH_ON() && !PUMP_FAIL()) {
        @@ Assert SwitchOn=true & PumpFail=false;
        @@ Update MC=Operating;
      }
      else {
        READ_DEVICE();    /* read state of the pump */
        @@ Update PumpFail=Top;
        if (PUMP_FAIL()) {
          @@ Assert PumpFail=true;
          @@ Update MC=Error & PumpOn=false;
        }
      }
    }
    else if (IN_OPERATING(System)) {
      @@ Assert MC=Operating;
      READ_SWITCH();
      @@ Update Switch=Top;
      if (!SWITCH_ON() && !PUMP_FAIL()) {
        @@ Assert SwitchOn=false & PumpFail=false;
        @@ Update MC=Off;
      }
      else {
        GET_WATER_LEVEL (&Water);    /* compute water level */
        @@ Update TooHigh=Top & TooLow=Top;
        if (IS_HIGH(Water) || IS_LOW(Water)) {
          @@ Assert TooHigh=true | TooLow=true;
          @@ Update PumpOn=true;
        }
        else {
          @@ Assert TooHigh=false & TooLow=false;
          @@ Update PumpOn=false;
        }
        READ_DEVICE();
        @@ Update PumpFail=Top;
        if (PUMP_FAIL()) {
          @@ Assert PumpFail=true;
          @@ Update MC=Error & PumpOn=false;
        }
      }
    }
  }
}
```

Figure 3.2: Initial SWLMS design.

We are interested in extending our analysis technique to checking the consistency of

```
{ ...
  @@ Initial MC=Off & SwitchOn=false & PumpFail=false &
     TooHigh=false & TooLow=false & PumpOn=false;
  while(1) {
    READ_DEVICE();
    @@ Update PumpFail=Top;
    if (PUMP_FAIL()) {
      @@ Assert PumpFail=true;
      break;
    }
    @@ Assert PumpFail=false; /* assume no device failures */
    READ_SWITCH();            /* read switch monitor */
    @@ Update SwitchOn=Top;
    if (SWITCH_ON() && IN_OFF(System)) {
      @@ Assert SwitchOn=true & MC=Off;
      @@ Update MC=Operating;
    }
    else if (IN_OPERATING(System)) {
      @@ Assert MC=Operating;
      while(1) {
        READ_DEVICE();
        @@ Update PumpFail=Top;
        if (PUMP_FAIL()) {
          @@ Assert PumpFail=true;
          break;
        }
        @@ Assert PumpFail=false;
        READ_SWITCH();
        @@ Update SwitchOn=Top;
        if (!SWITCH_ON()) {
          @@ Assert SwitchOn=false;
          @@ Update MC=Off;
          break;
        }
        @@ Assert SwitchOn=true;
        GET_WATER_LEVEL (&Water); /* compute water level */
        @@ Update TooHigh=Top & TooLow=Top;
        if (IS_HIGH(Water) || IS_LOW(Water)) {
          @@ Assert TooHigh=true | TooLow=true;
          @@ Update PumpOn=true;
        }
        else {
          @@ Assert TooHigh=false & TooLow=false;
          @@ Update PumpOn=false;
        }
      }
    }
  }
  @@ Assert PumpFail=true;
  @@ Update MC=Error & PumpOn=false;}
```

Figure 3.3: Final SWLMS design.

designs and their implementations.

3.1.3 Discussion

A central notion of SCR requirements is that of an event - a change of value of a condition. In designs, events may occur when a condition is assigned some value, or when the condition's value is determined. For example, a design might read a sensor, but only when the water becomes too low should the pump be turned on. One of the possible ways to write designs for systems specified by SCR requirements is as shown in Figure 3.4. Events are defined explicitly by keeping previous and current values

```
while (1) {
  for each sensor {
     copy previous value of corresponding variable
     read the sensor
     determine if the event occurred by comparing old and new values
     determine state changes
     change values of controlled variables
  }
}
```

Figure 3.4: Possible way to write SCR-style designs.

of variables. Note that designs can be written this way only if we assume that no simultaneous events can occur. In the rest of the dissertation we assume that designs do not keep track of previous values of conditions.

3.2 Creating a Design Flow Graph

3.2.1 Overview of the Approach

We construct a Design-Flow Graph (DFG) from annotations and control-flow information of the design and then propagate the state information throughout the DFG, in a manner similar to data-flow analysis[ASU88]. The rest of this section describes the process we use to construct a set-based approximation of attainable values of requirements variables for each node of a DFG. The process consists of the following steps:

- Compute information generated by each annotation.
- Detail this information using environmental assumptions and check for violations of the assumptions.
- Propagate this information throughout the DFG.

Definition 3.1 *A Control-Flow Graph of a program design PD (DFG) is a directed graph* $G = \langle V,\ E,\ V_0 \rangle$, *where*

> V *is a finite set of nodes corresponding to splits, joins and annotations of PD.*
>
> $E \subseteq V \times V$ *is a set of directed edges, s.t.* $(v_1,\ v_2) \in E$ *iff* v_2 *can immediately follow* v_1 *in some execution sequence; and*
>
> $V_0 \in V$ *is an entry node.*

3.2.2 Set-Based Approach

We interpret annotations in the design to create a set-based approximation of attainable values for each requirements' variable at each node of the DFG[CC76, CC77a]. This means that we are interested in only those variables which are specified in the requirements, and that each variable is associated with a set of values it may attain if the control reaches that node.

Definition 3.2 *A system state at node* n *of a design implementing an SCR specification* $\mathcal{R}$ *is a set of variable-value pairs* $\{(r_j, v_{r_j}) \mid r_j \in R\}$, *where*

> v_{r_j} *is a set of values associated with the variable* r_j *at the node* n, *and*
>
> $\forall\ r_j \in R,\ v_{r_j} \in 2^{T(r_j)}$ *(*v_{r_j} *is a set consisting of values in the domain of* r_j*).*

We require that there is exactly one variable-value pair for each variable in the requirements, and thus for a system state s we can define a function $v(r_j, s)$ which returns the value of r_j in s:

$$v(r_j, s) \equiv \mu \text{ s.t. } (r_j, \mu) \in s$$

In addition, we define a function $\text{repl}(r_j, \mu, s)$ to replace the current value of r_j by μ in s:

$$\text{repl}(r_j, \mu, s) \equiv s - \{(r_j, v(r_j, s))\} \cup \{(r_j, \mu)\}$$

For controlled and monitored variables ($r_j \in M \cup C$), these values are $\{\}$, $\{\text{true}\}$, $\{\text{false}\}$ and $\{\text{true, false}\}$, which form a $\cup$-lattice on set-inclusion. These values have the following meaning:

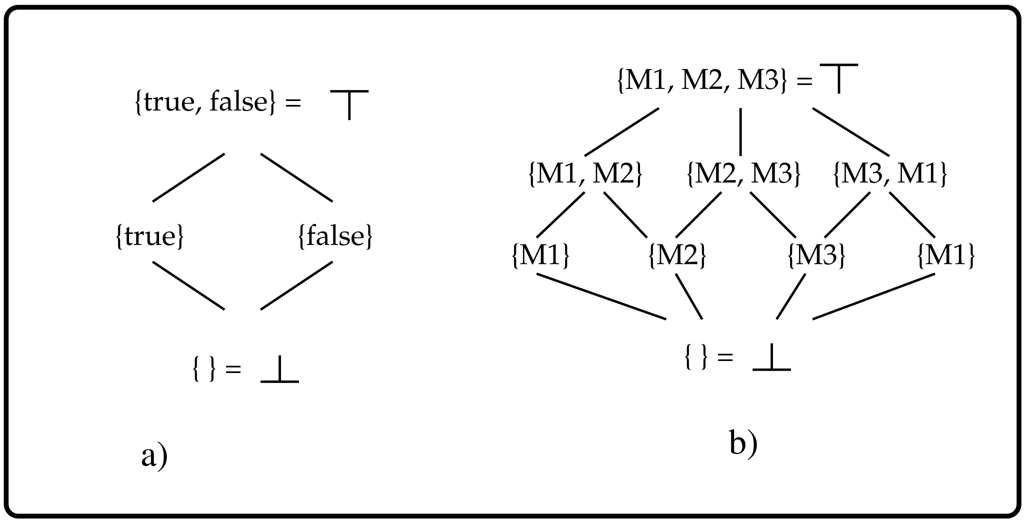


Figure 3.5: Lattice structure of requirements variables .

{ }	On any path leading to this node, the value of the variable is unknown.
{true}	On all paths leading to this node, the variable is true.
{false}	On all paths leading to this node, the variable is false.
{true, false}	On some path the variable is true, and on some other it is false.

For modeclasses ($r_j \in MD$), the values also form a $\cup$-lattice on set-inclusion. Figure 3.5a depicts a lattice of binary values, and Figure 3.5b depicts a lattice of values for a modeclass with modes M_1, M_2 and M_3. This lattice is three-dimensional, so we have to duplicate a few nodes in the two-dimensional presentation to depict the relationship between the nodes.

3.2.3 Operations on System States

Operations on system states are: "$\sqcup$" (union), "$\sqcap$" (intersection), "$=$" (equality), "$\sqsupset$" (superset), "$-$" (difference) and "$\sqsupseteq$" (superset or equal to). We use different symbols to distinguish these from operations on sets (except for "$-$" and "$=$", which might require an extra clarification of their meaning). Operations between system states ss_1 and ss_2 are defined in Figure 3.6. We also define a special system state EMPTY:

$$s = \text{EMPTY} \equiv \forall r_j \in R, v(r_j, s) = \{\}$$

System states form a complete $\sqcup$-lattice under the partial ordering of inclusion $\sqsupset$. Figure 3.7 depicts such a lattice for one monitored variable a and one modeclass MC with modes M_1, M_2 and M_3. Arrows and circled nodes indicate a possible path through the lattice. The height of the lattice, $\mathcal{H}_{inexact}$, grows proportionally to the sum of the

$$
\begin{aligned}
ss_1 \sqcup ss_2 &\equiv \{(r_j, \mu_j) \mid \forall r_j \in R, \mu_j = v(r_j, ss_1) \cup v(r_j, ss_2)\} \\
ss_1 \sqcap ss_2 &\equiv \{(r_j, \mu_j) \mid \forall r_j \in R, \mu_j = v(r_j, ss_1) \cap v(r_j, ss_2)\} \\
ss_1 - ss_2 &\equiv \{(r_j, \mu_j) \mid \forall r_j \in R, \mu_j = v(r_j, ss_1) - v(r_j, ss_2)\} \\
ss_1 = ss_2 &\equiv \forall r_j \in R, v(r_j, ss_1) = v(r_j, ss_2) \\
ss_1 \sqsupseteq ss_2 &\equiv \forall r_j \in R, v(r_j, ss_1) \supseteq v(r_j, ss_2) \\
ss_1 \sqsupset ss_2 &\equiv (ss_1 \sqsupseteq ss_2) \ \wedge \ (ss_1 \neq ss_2) \\
\mathrm{repl}(ss_1, ss_2) &\equiv \text{foreach } r_j \in R, \text{ if } v(r_j, ss_2) \neq \{\} \\
& \ \mathrm{repl}(r_j, v(r_j, ss_2), ss_1)
\end{aligned}
$$

Figure 3.6: Operations on system states for set-based approximation.

number of values each requirements variable can attain, i.e.

$$\mathcal{H}_{inexact} = \Sigma_{r_j \in R} |T(r_j)| + 1$$

For the lattice in Figure 3.7,

$$\mathcal{H}_{inexact} = 2 + 3 + 1 = 6$$

3.2.4 Constant Propagation

Our computation of system states at each node of the DFG is similar to that of constant propagation - a compiler technique whose goal is to discover values that are constant for all possible executions of a program and to propagate these constant values as far forward through the program as possible[WZ91, ASU88]. For every node n in

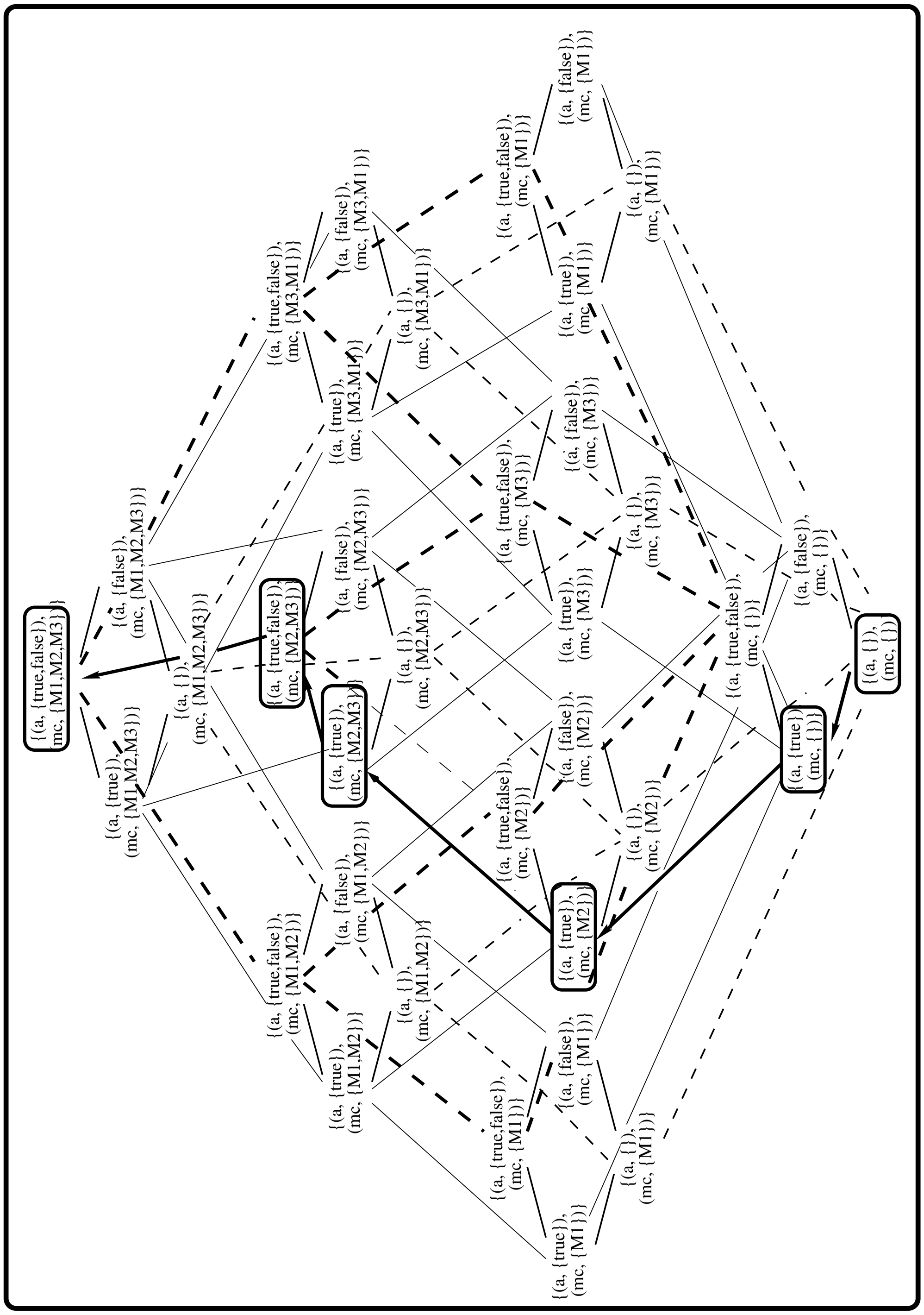


Figure 3.7: A lattice of system states. States with (mc, {M1})) have been duplicated for clarity.

the graph, we keep the following sets of variable-value pairs:

`gen`(n)	Pairs with values generated in the annotation at node n.
`known`(n)	Pairs with values assumed by the designer at node n.
`in`(n)	Pairs that may exist when control reaches n.
`out`(n)	Pairs that may exist when control leaves n.

3.2.5 Computing `gen` and `known` Sets

`gen` and `known` sets for each node are computed using the following rules:

- For nodes corresponding to Update and Initial annotations, `gen` sets contain variable-value pairs with the specified value, and with empty set values for all other variables.
- For nodes corresponding to Assert annotations, `known` sets contain variable-value pairs with the specified value, and with empty set values for all other variables.
- For all other nodes, `gen` and `known` sets are EMPTY.

Assert annotations may contain a disjunction of several clauses. For these nodes, `known` sets are lists of several system states, one for each clause. Operations on system states are trivially extended to handle sets of system states.

Once the initial `gen` and `known` sets are constructed, we use environmental assumption information (i.e., the $\mathcal{E}$ part of the SCR requirements) to make these sets more precise. For each node n, we change values of variables in `gen`(n) and `known`(n) sets based on the tables below. For brevity, we say repl$(r,\ \mu)$ in place of repl$(r,\ \mu,$ `gen`$(n))$ or repl$(r,\ \mu,$ `known`$(n))$. While detailing `gen` and known sets, we also check that environmental assumptions are preserved (property "ENV"). The error indicates a violation of ENV properties, i.e., that the user has inconsistently set several related variables in one Update annotation. For a number of environmental assumptions, we could deduce much more information than we actually do. However, we limit the use of these assumptions to detailing `gen` and `known` sets.

equality $(a = b)$

$a \backslash b$	{}	{true}	{false}	{true,false}
{}	no change	repl(a, {true})	repl(a, {false})	repl(a, {true,false})
{true}	repl(b, {true})	no change	error	repl(b, {true})
{false}	repl(b, {false})	error	no change	repl(b, {false})
{true,false}	repl(b, {true,false})	repl(a, {true})	repl(a, {false})	no change

implication $(a\text{–>}\ b)$, **strict implication** $(a\text{–>>}\ b)$ and **timeline** $(b < a)$

$a \backslash b$	{}	{true}	{false}	{true,false}
{}	no change	no change	repl(a, {false})	no change
{true}	repl(b, {true})	no change	error	repl(b, {true})
{false}	no change	no change	no change	no change
{true,false}	no change	no change	repl(a, {false})	no change

enumeration $(a_1 \mid a_2 \mid ... \mid a_k)$ In the table below, $i \neq j$.

$a_i \backslash a_j$	{}	{true}	{false}	{true,false}
{}	no change	repl(a_i, {false})	no change	no change
{true}	repl(a_j, {false})	error	no change	repl(a_j, {false})
{false}	no change	no change	no change	no change
{true,false}	no change	repl(a_i, {false})	no change	error

Examples:

In the examples below, we omit variable-value pairs in which the value is {}. For an annotation @@Update MC=Operating & PumpFail=true,

$$\texttt{gen} = \{(\text{MC}, \{\text{Operating}\}), (\text{PumpFail}, \{\text{true}\})\}$$

For an annotation @@Assert SwitchOn=false | PumpFail=true,

$$\texttt{known} = \{\{(\text{SwitchOn}, \{\text{false}\})\}, \{(\text{PumpFail}, \{\text{true}\})\}\}$$

Environmental assumption TooLow –>> ¬TooHigh is used to make the `known` set for an annotation @@Assert TooLow=true more precise. The resulting `known` set is

$$\texttt{known} = \{(\text{TooLow}, \{\text{true}\}), (\text{TooHigh}, \{\text{false}\})\}$$

3.2.6 Propagating Information

We initialize `in` and `out` sets of every node to EMPTY. Then we propagate information throughout the DFG until a least fixed point is reached. A <u>meet</u> operator for combining information coming into the node is $\sqcup$, so

$$\texttt{in}(n) = \sqcup_{\forall\, k,\, \text{s.t.}\, (k,n) \in E}\texttt{out}(k)$$

A set F of <u>transfer functions</u> describing the transformation between `in` and `out` sets at each node, is defined as follows:

Annotation at node n	Transfer function
Initial	$\texttt{out}(\text{n}) = \texttt{gen}(\text{n})$
Update	$\texttt{out}(\text{n}) = \text{repl}(\texttt{in}(\text{n}), \texttt{gen}(\text{n}))$
Assert (single disjunct)	$\texttt{out}(\text{n}) = \texttt{in}(\text{n}) \sqcap \texttt{known}(\text{n})$
none	$\texttt{out}(\text{n}) = \texttt{in}(\text{n})$

If the $\texttt{known}$ set for a node n containing an Assert annotation consists of several disjuncts, i.e., $\texttt{known}(n) = \{d_1, d_2, ..., d_k\}$, then

$$\texttt{out}(n) = \sqcup_{1 \le i \le k}(\texttt{in}(n) \sqcap d_i)$$

Our framework is strictly monotonic, i.e., $\texttt{in}$ and $\texttt{out}$ sets on $n+1$st iteration of our algorithm have more values (or at least as many) for each variable as $\texttt{in}$ and $\texttt{out}$ sets on nth iteration. Since all variables in R have a finite number of abstract values, our system states do not have an infinite increasing chain of values, and the fixed point can be achieved in a finite number of steps. The height of the lattice of system states is the length of the longest increasing chain of values, i.e., the maximum number of times that information for each node can be changed before the fixed point is achieved is

$$\mathcal{H} = \Sigma_{r_j \in R} |T(r_j)|.$$

Thus, the entire computation of the DFG can take at most

$$O(|V|^2 \times \Sigma_{r_j \in R} |T(r_j)|)$$

steps, where $|V|$ is the number of nodes in the DFG.

If the system could have an infinite increasing chain of values, the fixed point could not be achieved in a finite number of steps using data-flow techniques. This problem can be solved using another technique, abstract interpretation[CC76, CC77b].

We assume that all variables are initialized by the Initial annotation.

Definition 3.3 *A node n is considered unreachable if*

$$\exists r_j \in R \ s.t. \ v(r_j, \texttt{out}(n)) = \{\}$$

3.2.7 Example - DFG of SWLMS

Consider computing DFG for the design of SWLMS (Figure 3.3). Figure 3.8 shows a fragment of this DFG corresponding to lines 23-30 of the SWLMS design:

```
if (IS_HIGH(Water) || IS_LOW(Water)) {
  @@ Assert TooHigh=true | TooLow=true;
  @@ Update PumpOn=true;
}
else {
  @@ Assert TooHigh=false & TooLow=false;
  @@ Update PumpOn=false;
}
```

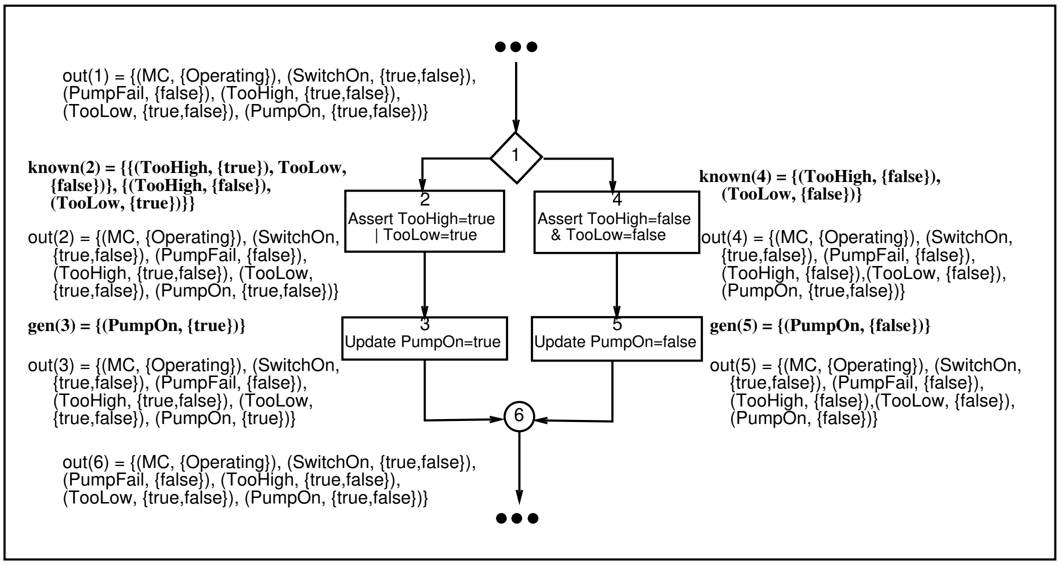


Figure 3.8: A fragment of DFG of SWLMS.

`gen` and `known` sets computed at each node are shown in bold font in this figure; variables whose values are {} are omitted. The Assert on the left branch (node 2) generates information that either TooLow or TooHigh is true. If TooLow is true, then, via environmental restrictions, TooHigh is false, and vice versa. The resulting `known` set consists of two disjuncts, one for each possibility. The Update on that branch (node 3) generates the value {true} for PumpOn. The Assert on the right branch (node 4) generates the value {false} for TooLow and TooHigh, and the following Update (node 5) generates the value {false} for PumpOn. In Figure 3.8, `out` sets are shown in regular font. We did not include `in` sets in this figure, since the `in` sets for nodes 2-5 are equal to the `out` sets of their predecessor nodes; and the `in` sets for nodes 1 and 6 are equal to their `out` sets. To compute `out` sets, Analyzer uses the multiple disjunct transfer function for Assert nodes. Each of disjuncts of `known`(2) is intersected with `in`(1), and the results are unioned, resulting in values of {true, false} for both TooHigh and TooLow in `out`(2). The Update annotation PumpOn=true (node 3) changes the value of PumpOn in `out`(3) to {true}. The right branch is processed similarly. At the join, we union the possible values for each variable in the `out` sets of the predecessor nodes (nodes 3 and 5).

3.3 An Alternative Approach - Exact Analysis

3.3.1 Motivation

The set-based approximation used in the previous section has the disadvantage of losing inter-variable dependencies. Set-based approximation applied to the following fragment of a (modified) SWLMS design:

```
...
@@ Assert TooLow=false & PumpOn=false;
if (...) {
   @@ Update TooLow=false;
   @@ Update PumpOn=true;
}
else {
   @@ Update TooLow=true;
   @@ Update PumpOn=false;
}
...
```

results in `out`={(PumpOn, {true, false}), (TooLow, {true, false})} at the join node. Thus, we lose information that the value {true} of PumpOn is associated with the value {false} of TooLow, and the value {false} of PumpOn is associated with the value {true} of TooLow. An approach that keeps track of inter-variable dependencies is called exact analysis. Exact analysis has been implemented in another version of Analyzer. The rest of this section describes this analysis technique.

3.3.2 Operations on System States

Under exact analysis, each system state consists of a set of state instances.

Definition 3.4 *A state instance at a node n of a design implementing an SCR specification $\mathcal{R}$ is a set of variable-value pairs $\{(r_j, v_{r_j})\}$ where*

v_{r_j} is a value associated with the variable r_j at the node n, and

$\forall\ r_j \in R$, $(v_{r_j} \in T(r_j)) \vee (v_{r_j} = \text{UNDEF})$.

Again, we assume that each state instance si has exactly one variable-value pair for each variable in R and define a function $v(r_j, si)$, returning r_j's value in si:

$$v(r_j, si) \equiv \mu \text{ s.t. } (r_j, \mu) \in si$$

Each state instance at node n represents a snapshot of values of all variables for some path leading to n, and each system state, a set of state instances, represents information gathered along all paths leading to n. Undefined variables are marked with a special value UNDEF. Thus, for boolean values we have three-valued logic, with values UNDEF, true and false. For the example presented in Section 3.3.1, the system state `out` at the join node is {{(TooLow, false), (PumpOn, true)}, {(TooLow, true), (PumpOn, false)}}.

Definition 3.5 *A node n is considered* <u>*unreachable*</u> *if some variable is UNDEF in all state instances of* `out`*(n).*

Operations on system states: "+" (union) and "∩" (intersection) are defined in Figure 3.9. This figure also describes the function repl to replace the value of a variable in a state instance and system state. In this figure, μ_1 and μ_2 are values of variables, si_1 and si_2 are state instances, and ss_1 and ss_2 are system states. Note that "+" is not a commutative operation, i.e., in definition of $\mu_1 + \mu_2$, the second variable affects the value of the first only if the first variable is undefined. The remaining operations, like subset and equality, are the same as those used in set-based approximations.

3.3.3 Computing `ins` and `outs`

`gen` and `known` sets are {} for all nodes except those corresponding to annotations. Variables not present in the annotation are associated with the value UNDEF. Values of variables in the annotations are used to form variable-value pairs for `gen`s. If more than one value for each variable needs to be represented, several state instances are created, one for each value. For example, annotations

```
1: @@ Update MC=Off & PumpOn=false;
2: @@ Assert ~(MC=Off) | PumpOn=true;
3: @@ Assert MC=Off & PumpOn=Top;
```

in the design of SWLMS, result in the sets below. The number in parenthesis indicates the line corresponding to the set. Variables whose values are UNDEF are not shown in the sets.

$$\mu_1 + \mu_2 \equiv \text{if } (\mu_1 = \text{ UNDEF }) \text{ then } \mu_2 \text{ else } \mu_1$$

$$\mu_1 \cap \mu_2 \equiv \text{if } (\mu_1 = \mu_2) \text{ then } \mu_1 \text{ else UNDEF}$$

$$\begin{aligned}
si_1 + si_2 &\equiv \{(r_j, \mu_j) \mid \forall r_j \in R, \mu_j = v(r_j, si_1) + v(r_j, si_2)\} \\
si_1 \cap si_2 &\equiv \{(r_j, \mu_j) \mid \forall r_j \in R, \mu_j = v(r_j, si_1) \cap v(r_j, si_2)\} \\
\text{repl}(r_j, \mu, si) &\equiv (si - \{(r_j, v(r_j, si))\} \cup \{(r_j, \mu)\} \\
\text{repl}(si_1, si_2) &\equiv \text{foreach } r_j \in R, \text{ if } v(r_j, si_2) \neq \text{UNDEF} \\
&\quad\ \text{repl}(r_j, v(r_j, si_2), si_1)
\end{aligned}$$

$$\begin{aligned}
ss_1 + ss_2 &\equiv \{(si_1 + si_2) \mid \forall si_1 \in ss_1 \wedge \forall si_2 \in ss_2\} \\
ss_1 \cap ss_2 &\equiv \{(si_1 \cap si_2) \mid \forall si_1 \in ss_1 \wedge \forall si_2 \in ss_2\} \\
\text{repl}(ss, si) &\equiv \{\text{repl}(si\prime, si)\ \forall si\prime \in ss\} \\
\text{repl}(ss_1, \{si_1, si_2, \cdots, si_k\}) &\equiv \text{repl}(ss_1, si_1) + \text{repl}(ss_1, si_2) + \cdots + \text{repl}(ss_1, si_k)
\end{aligned}$$

Figure 3.9: Operations on system states for exact analysis.

$$\begin{aligned}
\texttt{gen}(1) &= \{\{(\text{MC, Off}), (\text{PumpOn, false})\}\} \\
\texttt{known}(2) &= \{\{(\text{MC, Operating})\}, \{(\text{MC, Error})\}, \{(\text{PumpOn, true})\}\} \\
\texttt{known}(3) &= \{\{(\text{MC, Off}), (\text{PumpOn, true})\}, \{(\text{MC, Off}), (\text{PumpOn, false})\}\}
\end{aligned}$$

`in` and `out` sets are initialized to {} and `gen` and `known` sets are propagated throughout the DFG via a least fixed point algorithm.

$$\texttt{in}(n) = \bigcup_{\forall\, k,\ \text{s.t. } (k,n) \in E} \texttt{out}(k)$$

A set F of transfer functions is defined as follows:

Annotation at node n	Transfer function
Initial	`out`(n) = `gen`(n)
Update	`out`(n) = repl(`in`(n), `gen`(n))
Assert	`out`(n) = `in`(n) $\cap$ `known`(n)
none	`out`(n) = `in`(n)

These equations are very similar to the ones in Section 3.2.6. However, the top of the lattice of system states under exact analysis is a set containing all possible snapshots of the system. Thus the height of this lattice, $\mathcal{H}_{exact}$, is

$$\mathcal{H}_{exact} = 3^{|M \cup C|} \times \Pi_{1 \leq j \leq |MD|}(|T(md_j)| + 1),$$

since in addition to all the usual values, each variable can be UNDEF. In this formula, $|M|$ is a number of monitored variables in the SCR specification, $|C|$ is a number of controlled variables, $|MD|$ is a number of modeclasses, and $|md_j|$ is a number of modes in the j'th modeclass. Environmental assumptions can make $\mathcal{H}_{exact}$ smaller than the number above, although it is still much larger than the height of the lattice in the set-based approach. In the worst case, the number of steps taken by the entire computation is

$$O(|V|^2 \times 3^{|M \cup C|} \times \Pi_{1 \leq j \leq |MD|}(|T(md_j)| + 1)).$$

The following table lists heights of lattices for exact analysis and set-based approximation, $\mathcal{H}_{exact}$ and $\mathcal{H}_{inexact}$ for problems of various size. Numbers in parentheses indicate the number of modes in each modeclass, e.g. 2 (4, 3) indicates that the problem consists of two modeclasses, with 4 and 3 modes each. The variables in specification of A-7E aircraft[AFB+88] refer to just monitored variables.

Boolean variables	5	10	25	50	69
Modeclasses	1 (3)	2 (2, 3)	2 (4, 3)	3 (4, 5, 6)	3 (6, 17, 18)
Comments	SWLMS		WLMS (Chapter 5)		A-7 Spec
$\mathcal{H}_{exact}$	729	708588	1.69×10^{13}	1.5×10^{26}	2.0×10^{36}
$\mathcal{H}_{inexact}$	14	26	58	116	180

3.3.4 Analysis

Exact analysis promises dramatic improvement in quality of the analysis. However, due to the large size of system states, we need to use more sophisticated techniques to be able to analyze reasonable-size systems. During analysis of SWLMS, we did not uncover any new problems and there was no noticeable difference in response time between exact and inexact analysis. However, when we tried analyzing the Water-Level Monitoring System (a system with 18 monitored and 7 controlled variables and two modeclasses with 4 and 3 modes) using exact analysis, the current version of Analyzer took more than an hour, as opposed to several seconds for the set-based approximation.

In some cases, like the one presented below,

```
...
@@ Assert TooLow=false & PumpOn=false;
if (...)
   @@ Update TooLow=false;
else
   @@ Update TooLow=true;
if (...)
   @@ Update PumpOn=true;
else
   @@ Update PumpOn=false;
...
```

exact analysis makes states larger without yielding any extra information. At the end of the second If statement, the system state is

$$\{\{(\text{TooLow, false}),\ (\text{PumpOn, true})\},\ \{(\text{TooLow, false}),\ (\text{PumpOn, false})\}$$
$$\{(\text{TooLow, true}),\ (\text{PumpOn, true})\},\ \{(\text{TooLow, true}),\ (\text{PumpOn, false})\}\}$$

The right approach might be to combine the two types of analysis into a hybrid technique, where certain variables are to be analyzed exactly, and the rest approximately. The rest of this dissertation describes set-based approximation approach.

3.4 Processing Functions

In Section 3.2, we presented our algorithm for intraprocedural analysis of designs, which results in the creation of a DFG. However, since designs usually consist of several procedures, we need to extend our analysis technique to process these designs. We perform interprocedural analysis using an adaptation of a technique called cloning [CHK92]. A similar algorithm was described in [Weg75]. This technique enables Analyzer to analyze each called function only a constant number of times, and to achieve reasonable precision in the analysis. Our algorithm clones a control-flow graph (DFG) of a design of a function each time it is called in a new calling context. We verify each copy of a function once with its own set of reaching values and conditions. Since we ignore functions without annotations and the number of possible calling contexts is finite (and hopefully small), we avoid many potential problems with the combinatorial growth of the DFG of the entire program.

Definition 3.6 *A DFG of a program design PD in the presence of function calls is a directed graph* $G = \langle V,\ E,\ V_0\rangle$, *where*

V is a finite set of nodes containing control-flow constructs (decisions, joins, function calls) and annotations of PD.

$E \subseteq V \times V$ is a set of directed edges, s.t. $(v_1, v_2) \in E \leftrightarrow v_2$ can immediately follow v_1 in some execution sequence; and

V_0 is an entry node.

3.4.1 Algorithm

We store separate copies of DFGs of each function for different calling contexts. Every call site contains a pointer to a copy of a DFG of the called function, or a null pointer if it calls a function without annotations. `first`(f) and `last`(f) are pointers to the first and the last block of function f. To distinguish between copies of functions, we associate a list of contexts with each of them. Let `contexts`(f) be a list of contexts for a function f. Each context $ct \in$ `contexts`(f) contains the following information:

`in`(ct)	Information at the beginning of the function.
`out`(ct)	Information at the end of the function.
`graph`(ct)	Pointer to the DFG of the function with this calling environment.
`color`(ct)	Processing status: WHITE (unprocessed), GRAY (partially processed), or BLACK (processed).

Initially, `ins` and `outs` are EMPTY, and each context is marked WHITE. Let $\mathcal{F}$ be a set of functions in the design which needs to be processed, i.e. those which contain annotations or call other functions which need to be processed. The algorithm is shown in Figure 3.10. In the presentation of the algorithm, CopyDFG(f) is a function that takes a pointer to a function f and returns a pointer to a copy of f. The algorithm works as follows: when a function call is encountered, we check if this function has been

Algorithm 1

Inputs: A set of DFGs for each function.
Outputs: A combined DFG of the entire design.
Algorithm:

```
start computing in and out sets starting with Main function
(as in the interprocedural case).
if node n calls a function f ∈ F then {
   if ∃ct ∈ contexts(f) s.t. in(ct) = in(n) then {
      if color(ct) is BLACK (already processed) then {
         graph(n) := graph(ct)
         out(n) := out(ct)
      }
      else if color(ct) is GRAY then { /* recursion */
         report error (recursion is not supported)
         graph(n) := graph(ct)
         out(n) := in(ct) /* skip the call */
      }
   }
   else { /* no context was found */
      nct := new context
      graph(nct) := CopyCFG(graph(f))
      graph(n) := graph(nct)
      color(nct) := GRAY
      in(nct) := in(first(f)) := in(n)
      Compute in and out sets for the new copy of f
      color(nct) := BLACK
      out(last(f)) := out(n) := out(nct)
      add nct to contexts(f)
   }
}
```

Figure 3.10: Algorithm to process function calls.

called in the same environment (i.e., with the same `in` information). If so, we point to the body of the function in this environment. If the function has not been called in this environment, we inline the function and compute `in` and `out` sets for it.

3.5 Constructing a Finite-State Machine

The DFG that we create contains nodes which do not reflect state changes (e.g., decision nodes, joins and nodes containing Assert annotations) and possibly some unreachable nodes. We construct a Finite-State Machine (FSM) which contains just reachable nodes representing state changes. The resulting FSM is used as a model for verifying system properties. In a DFG ($G = \langle V, E, V_0 \rangle$), let $U \subseteq V$ and $I \subseteq V$ be sets of nodes containing Update and Initial annotations, respectively. (U and I are disjoint.)

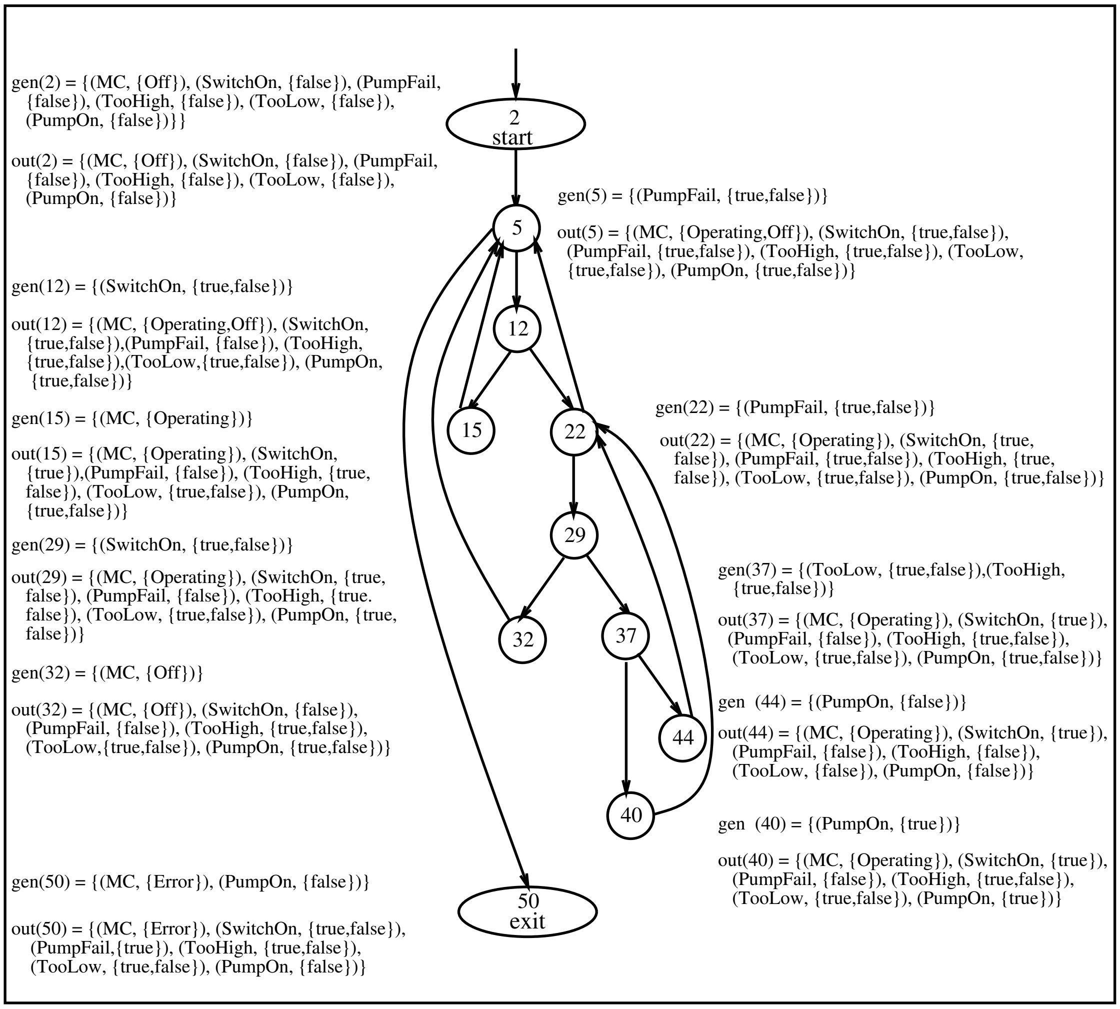


Figure 3.11: Finite-state abstraction.

Definition 3.7 *A Finite-State Machine (FSM) over a program design PD is a structure $M = \langle A,\ S,\ L,\ N,\ s_0 \rangle$, where*

A is a set of labels;

$S = U \cup I$ is a finite set of nodes;

L: $S \rightarrow A$ is a function associating each node with a label;

$N \subseteq S \times S$ is a transition relation.

N is obtained by connecting nodes of S s.t. there is an Update-clear path between them in DFG; and

$s_0 \in S$ is an entry node.

To build a FSM from a DFG we remove all nodes except those corresponding to Initial and Update annotations and connect all predecessors of a removed node to the node's successors. For every node, we check an implicit property (REACH):

$$\forall n \in S(\forall r_j \in R,\ v(r_j, out(n)) \neq \{\})$$

If this property is violated in a node, an error is reported and the node is removed from the FSM. The running time to build the FSM is proportional to $|V|$, and the number of nodes in the resulting FSM is bounded above by $|S|$, the number of Update and Initial statements in the design.

Figure 3.11 shows the FSM created for the SWLMS design in Figure 3.3, depicting `out` and `gen` sets for every node. The number of each node of the FSM indicates the line of the design at which the corresponding Update or Initial annotation can be found. For example, nodes 40 and 44 correspond @@Update PumpOn=true and @@Update PumpOn=false, respectively. The algorithm for computing `out` sets described in Section 3.2.6 ensures that the effects of Assert annotations are preserved in system states, even though Assert nodes themselves are removed.

All states in the resulting FSM correspond to state changes. This FSM is then used to verify properties generated from requirements.

Chapter 4

Verifying Properties

During the process described in the previous chapter, we took a detailed design and built a FSM from it, combining all possible values of each variable at every program point and removing nodes which do not correspond to state changes or which are clearly unreachable. In this chapter we describe how to use the FSM to verify properties generated from SCR specifications. Verification is done by a model-checking algorithm.

4.1 Overview of Model-Checking

Introduced by Clarke and Emerson[CES86] and by Quielle and Sifakis[QS81], model checking determines the value of a formula f, typically expressed in temporal logic, for a particular system by building its reachability graph M and computing the set of states in which the formula is true, i.e., $\{s \mid M, s \models f\}$. The formula is considered to hold in the model if it holds in its initial state, i.e., $M, s_0 \models f$. Automating model checking is quite easy, and the resulting algorithm is linear both in the size of the formula and the number of nodes in the model. However, the entire state space of the model is constructed before the fixed point algorithms can be applied. Various improvements to the original technology allows for efficient checking of large systems[BCM+90, CGL92, McM93]. The problems successfully solved by model checking are mostly specifications of systems[Jac94, SA96, ABB+96] and hardware designs[CES86, CGH+93, BCDM86, MS92, Bar95]. Software systems are typically result in a very large state space, and thus some abstractions have to be used to analyze these systems[WVF95].

4.2 Motivation: Why Build Our Own Model-checker?

Our methods for constructing finite-state abstractions produced sets of values for each program variable. Set-based approximation stored such sets for every variable, and exact analysis kept snapshots of values of variables for each path to the given node. Available model checkers process states whose variables have scalar values. Transforming our FSM to correspond to an acceptable input for an existing model-checker would have resulted in an exponential increase in the number of nodes in the FSM. So, we developed our own technique to verify properties. We cannot verify properties exactly, i.e., claim that a property is violated if and only if we find a violation. So, we have carefully designed our verification algorithms so that the results can be correctly interpreted.

4.3 Optimism and Pessimism

We divide properties into two categories: those checked optimistically and pessimistically.

Definition 4.1 *A property is checked pessimistically if all of its violations in the design are detected, but the analysis incorrectly identifies violations at points in the design at which the property actually holds. A property is checked optimistically if all detected violations are present in the design, but the analysis is unable to find all violations of the property.*

Thus, if the analysis reports no violations of a pessimistic property P, then we know that P holds in the design; and if the analysis reports violations of an optimistic property P, then we know that P does not hold in the design. A property P is checked inconsistently if no valid conclusion can be made when a violation of P is reported or when no violations of P are reported. We want to avoid inconsistent checking by verifying each property either optimistically or pessimistically.

Let $\mathcal{P}$ be the set of properties which we are to verify for a model of a design. $\mathcal{P}$ consists of OLT ("only legal transitions"), ALT ("all legal transitions"), USER ("system goals are preserved"), ENV("environmental assumptions are preserved"), and REACH ("all nodes are reachable") property. The first three properties are generated from SCR requirements and are defined in Chapter 2. Some ENV properties are generated and checked during the DFG construction (Chapter 3). The remaining ENV properties are generated and checked during the process of computing events, as described in Section 4.5. Finally, the REACH property is generated and checked by the process of constructing the FSM, as described in Chapter 3.

4.4 Computing Transitions

Most of properties that we are interested in verifying involve state transitions, which occur only in response to events. Thus, we need to develop techniques to compute events and transitions for both optimistic and pessimistic analysis. Our techniques overestimate the number of transitions in the design, i.e., if a transition is present in the design, we compute it, but some of the computed transitions might not be present in the design. Overestimation of the number of transitions does not invalidate the verification of automatically-generated properties. For OLT properties, we want to check if all transitions in the design are present in the requirements, and overestimating the transitions might cause the tool to report some false negatives, which is a correct treatment of pessimistic analysis. For ALT properties, we want to check if all transitions in the requirements are present in the design, and overestimating the transitions might cause the tool to report some false positives, which is a correct treatment of optimistic analysis. Table 4.1 summarizes our analysis. For example, when a transition in the

Row	Transition exists			Analysis reports	
	Requirements	Computation	Design	ALT properties	OLT properties
1	T	T	T	no violation	no violation
2	T	T	F	false positive	no violation
3	T	F	T	–	–
4	T	F	F	violation	no violation
5	F	T	T	no violation	violation
6	F	T	F	no violation	false negative
7	F	F	T	–	–
8	F	F	F	no violation	no violation

Table 4.1: Results of analysis.

requirements is not implemented in the design and is not computed by our tool (row 4), then the tool reports a violation of the corresponding ALT property and does not report a violation of an OLT property. Our computation finds all transitions present in the design. Thus the cases described by rows 3 and 7 in Table 4.1 cannot occur, so the analysis results are not defined for them.

4.4.1 Mapping Between Events and Transitions in Requirements and Our Model

The semantics of an SCR event, given in Chapter 2, indicates that some of conditions need to hold on edges. However, our FSM consists of states, with `in`, `gen` and `out` sets. Pictorially, the mapping between events and transitions in the requirements and the FSM is shown in Figure 4.1. Formally, we say that a formula f holds in a state s if f

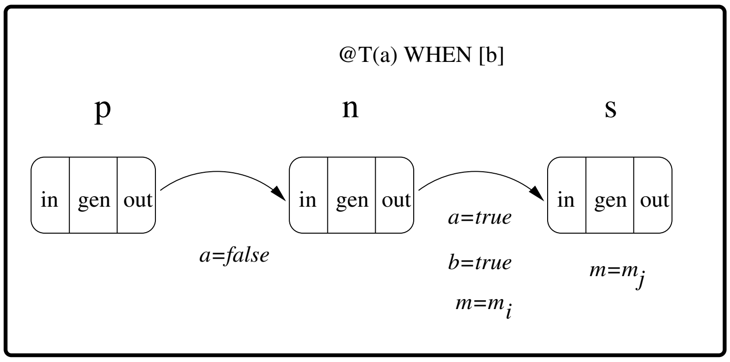


Figure 4.1: Pictorial representation for a mode transition.

holds in $\texttt{out}(s)$, i.e., $\texttt{out}(s) \models f$. We say that f is generated in a state s if $\texttt{gen}(s) \models f$. Finally, we say that f holds on an edge between nodes n and s if $(\texttt{out}(n) \sqcap \texttt{in}(s)) \models f$. By construction of the FSM, if a formula holds on exit of the node but does not hold on entrance, then it has been generated at this node, i.e. for a node n,

$$(\texttt{out}(n) \models f) \wedge (\texttt{in}(n) \not\models f) \rightarrow (\texttt{gen}(n) \models f)$$

We also note that an event occurs at a node if a value of some variable on an edge entering the node is different from its value on an edge leaving the node. Thus, the variable is changed in the node's `gen` set.

4.4.2 Algorithm to Compute Transitions

To determine if an event occurred at a node s, we use information generated at each predecessor node n and check it against `in`(s) and the `out` sets of the predecessors of n. The algorithm to compute transitions leading to a node s is shown in Figure 4.2. The output of this algorithm is a set of transitions $\{(n,$ `to`, `from`, `trigger`, `when`$)\}$, where

- n is a predecessor of s;
- `to` and `from` are in the form $r_j = v_{r_j'}$ and $r_j = v_{r_j}$, respectively, indicating the change of value of r_j from v_{r_j} to $v_{r_j'}$ for a transition between n and s;
- `trigger` is logical expression - a disjunction of simultaneously occurring triggering conditions, or NONE, if no events can occur; and
- `when` is a set of variable-value pairs indicating the When condition for this transition. We assume that controlled variables cannot be part of Triggering conditions, and that variables which are part of a Triggering condition cannot be part of a corresponding When condition.

The function events(r_j, s_1, s_2) checks values of a variable r_j in system states s_1 and s_2 to determine if an event involving r_j has occurred, returning a disjunction of @T(r_j), @F(r_j) or none(r_j). The last value indicates that no event involving r_j has occurred between the two nodes. We use "none" as a symbolic constant to indicate that some variable did not change its value. When the second system state refers to the initial node of the FSM, i.e., the node containing the Initial annotation, we assume that all values assigned to variables in this annotation signify events. For the initial node and

Algorithm 2

Inputs: Finite state machine (FSM) and
node s for which to compute the transitions.

Outputs: A set of transitions to node s. Each transition is in the form (n, `to`, `from`, `trigger`, `when`), where $n \in \text{pred}(s)$ and `to` $\neq$ `from`.

Algorithm:

```
all_transitions = {}
For each n ∈ pred(s) {
    /* triggs is a logical expression ⋀_{r_j ∈ R−C} ⋁_k exp_{j,k} */
    triggs = true
    when = out(n) ⊓ in(s)
    For each r_j ∈ R − C, where v(r_j, gen(n)) ≠ {}
        /* Controlled variables cannot be part of the triggering condition */
        triggs = triggs ∧ events(r_j, ⋃_{∀p∈pred(n)} out(p), gen(n) ⊓ when)
    /* Rewrite triggs to be in form ⋁_k ⋀_{r_j ∈ R} exp_{k,j} */
    For each of the k disjuncts d_k of triggs {
        Evaluate d_k interpreting none(r_j) as true.
        If d_k = true, then replace d_k with special value NONE
        trigger[k] = d_k
        when[k] = when
        /* remove trigger values from WHEN condition */
        For each r_j ∈ R − C
            If partof(r_j, d_k)
                when[k] = repl(r_j, when[k], {})
        /* compute to and from values for modeclass and controlled variables */
        For each r_j ∈ R − M s.t. v(r_j, gen(s)) ≠ {} {
            /* remove values changed between source and destination nodes */
            when[k] = repl(r_j, when[k], {})
            For each value_l ∈ v(r_j, gen(s)) {
                to = (r_j = value_l) /* final value of r_j */
                For each value_m ∈ v(r_j, out(n) ⊓ in(s)) {
                    from = (r_j = value_m) /* starting value of r_j */
                    If value_l ≠ value_m /* otherwise there is no transition */
                        all_transitions = all_transitions ∪
                                (n, to, from, trigger[k], when[k])
                }
            }
        }
    }
}
```

Figure 4.2: Algorithm for computing transitions.

---

a boolean variable r_j, events(r_j, $s_1 = \{\}$, s_2) is defined as

$v(r_j, s)$	events(r_j, {}, s)
{true}	@T(r_j)
{false}	@F(r_j)
{true,false}	@T(r_j) $\vee$ @F(r_j)

If the second system state refers to a non-initial node, then, for a boolean r_j, events(r_j, s_1, s_2) is defined as follows:

$v(r_j, s_1) \setminus v(r_j, s_2)$	{true}	{false}	{true,false}
{true}	none(r_j)	@F(r_j)	@T(r_j) $\vee$ none(r_j)
{false}	@T(r_j)	none(r_j)	@F(r_j) $\vee$ none(r_j)
{true,false}	@T(r_j) $\vee$ none(r_j)	@F(r_j) $\vee$ none(r_j)	@T(r_j) $\vee$ @F(r_j) $\vee$ none(r_j)

For a modeclass mc, events(mc, s_1, s_2) returns a disjunction of all possible event combinations which could occur between the two nodes. A combination of events is a conjunction of events which occur simultaneously. For example, if $v(mc, s_1)$ = {m1, m2} and $v(mc, s_2)$ = {m2, m3}, then

$$\begin{aligned}\text{events}(mc, s_1, s_2) \quad = \quad & ((\text{@T(mc=m2)} \wedge \text{@F(mc=m1)}) \;\vee\; \text{none(mc)} \;\vee \\ & (\text{@T(mc=m3)} \wedge \text{@F(mc=m1)}) \;\vee \\ & (\text{@T(mc=m3)} \wedge \text{@F(mc=m2)}))\end{aligned}$$

Finally, a function partof(k_l, `trigger`) returns true if `trigger` contains a conjunct corresponding to k_l and false otherwise.

We illustrate the algorithm for computing transitions using a fragment of the FSM shown in Figure 4.3. In this example, we are interested in computing transitions for node s. a and b are monitored variables, and m is a modeclass with modes m_1, m_2 and m_3. s has two predecessor nodes, n_1 and n_2. For node n_1,

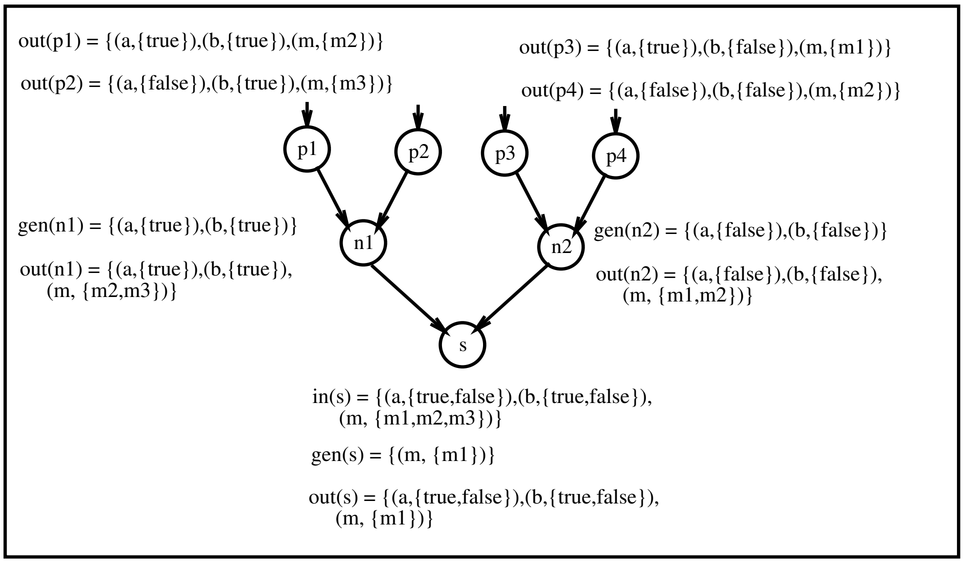


Figure 4.3: Fragment of FSM.

$$
\begin{aligned}
\texttt{when} &= \texttt{out}(n_1) \sqcap \texttt{in}(s) \\
&= \{(a,\{\text{true}\}),\ (b,\{\text{true}\}),\ (m,\{m_2,m_3\}) \sqcap \{(a,\{\text{true,false}\}), \\
&\quad (b,\{\text{true,false}\}),\ (m,\{m_1,m_2,m_3\})\} \\
&= \{(a,\{\text{true}\}),\ (b,\{\text{true}\}),\ (m,\{m_2,m_3\})\} \\
\texttt{out}(p_1) \sqcup \texttt{out}(p_2) &= \{(a,\{\text{true,false}\}),\ (b,\{\text{true}\}),(m,\{m_2,m_3\})\} \\
\texttt{gen}(n_1) \sqcap \texttt{when} &= \{(a,\{\text{true}\}),\ (b,\{\text{true}\})\})
\end{aligned}
$$

We loop over the variables a and b to compute changes in their values and accumulate the changes in triggs.

$$
\begin{aligned}
\text{triggs} &= \text{true} \wedge \\
&\quad \text{events}(a,\ \{(a,\{\text{true,false}\}),\ (b,\{\text{true}\}),(m,\{m_2,m_3\})\}, \\
&\quad \{(a,\{\text{true}\}),\ (b,\{\text{true}\})\}) \wedge \\
&\quad \text{events}(b,\ \{(a,\{\text{true,false}\}),\ (b,\{\text{true}\}),\ (m,\{m_2,m_3\})\}, \\
&\quad \{(a,\{\text{true}\}),\ (b,\{\text{true}\})\}) \\
&= (@\text{T}(a) \vee \text{none}(a)) \wedge \text{none}(b)
\end{aligned}
$$

We rewrite triggs into disjunctive normal form and evaluate each disjunct. The result is

$$
\begin{aligned}
\text{triggs} &= (@\mathrm{T}(a) \wedge \text{none}(b)) \vee (\text{none}(a) \wedge \text{none}(b)) \\
&= @\mathrm{T}(a) \vee \text{NONE}
\end{aligned}
$$

We separate each of the disjuncts of triggs into `trigger`, an array representing separate events. The two triggering events are

$$
\begin{aligned}
\texttt{trigger}[1] &= @\mathrm{T}(a) \\
\texttt{trigger}[2] &= \text{NONE}
\end{aligned}
$$

These events correspond to changes of values of a. a was {true} in p_1 and {true} in n_1, so for an edge (p_1, n_1) there is no event for a. a was {false} in p_2, so for an edge (p_2, n_1) the event is @T(a). The computed triggers indicate these events.

For each event in `triggers`, we calculate a unique WHEN condition (`when`[k]) by replacing values of variables which occur in `trigger`[k] by {}. For `trigger`[1], we replace the value of a by {} in `when`[1]. `trigger`[2], on the other hand, does not contain any variables, so `when`[2] remains unchanged:

$$
\begin{aligned}
\texttt{when}[1] &= \{(a,\{\}), (b,\{\text{true}\}), (m,\{m_2, m_3\})\} \\
\texttt{when}[2] &= \{(a,\{\text{true}\}), (b,\{\text{true}\}), (m,\{m_2, m_3\})\}
\end{aligned}
$$

Next we replace values of variables changed at the destination node with empty-set values, resulting in:

$$
\begin{aligned}
\texttt{when}[1] &= \{(a,\{\}), (b,\{\text{true}\}), (m,\{\})\} \\
\texttt{when}[2] &= \{(a,\{\text{true}\}), (b,\{\text{true}\}), (m,\{\})\}
\end{aligned}
$$

Finally, we compute the different combinations of value changes between nodes n_1 and s and associate events with each of them. The only vl_{to} is m_1. So for all transitions,

$$\texttt{to} = (m = m_1)$$

Since

$$v(m, \texttt{out}(n_1) \sqcap \texttt{in}(s)) = \{m_2, m_3\}$$

we get two values of `from`:

$\texttt{from} = (m = m_2)$ and
$\texttt{from} = (m = m_3)$

Finally, all_transitions contains the following transitions:

1 $\{(n_1,\ m = m_1,\ m = m_2,\ @\mathrm{T}(a),\ \{(a,\ \{\}),\ (b,\ \{\mathrm{true}\}),\ (m,\ \{\})\}),$
2 $(n_1,\ m = m_1,\ m = m_2,\ \mathrm{NONE},\ \{(a,\ \{\mathrm{true}\}),\ (b,\ \{\mathrm{true}\}),\ (m,\ \{\})\}),$
3 $(n_1,\ m = m_1,\ m = m_3,\ @\mathrm{T}(a),\ \{(a,\ \{\}),\ (b,\ \{\mathrm{true}\}),\ (m,\ \{\})\}),$
4 $(n_1,\ m = m_1,\ m = m_3,\ \mathrm{NONE},\ \{(a,\ \{\mathrm{true}\}),\ (b,\ \{\mathrm{true}\}),\ (m,\ \{\})\})\}$

The next iteration of the loop (over n_2) is done similarly, producing the remainder of the transitions. all_transitions also contains

5 $\{(n_2,\ m = m_1,\ m = m_1,\ @\mathrm{F}(a),\ \{(a,\ \{\}),\ (b,\ \{\mathrm{false}\}),\ (m,\ \{\})\}),$
6 $(n_2,\ m = m_1,\ m = m_1,\ \mathrm{NONE},\ \{(a,\ \{\mathrm{false}\}),\ (b,\ \{\mathrm{false}\}),\ (m,\ \{\})\}),$
7 $(n_2,\ m = m_1,\ m = m_2,\ @\mathrm{F}(a),\ \{(a,\ \{\}),\ (b,\ \{\mathrm{false}\}),\ (m,\ \{\})\}),$
8 $(n_2,\ m = m_1,\ m = m_2,\ \mathrm{NONE},\ \{(a,\ \{\mathrm{false}\}),\ (b,\ \{\mathrm{false}\}),\ (m,\ \{\})\})\}$

Transitions 1, 4 and 7 are not possible in the design, and transitions 5 and 6 are between the same mode $m = m_1$.

In our SWLMS example, consider calculating the event which causes MC to be set to Error at node 50 of the FSM in Figure 3.11. Figure 4.4 shows a fragment of this FSM. `when` is $\texttt{out}(5) \sqcap \texttt{in}(50)$, namely, {(SwitchOn, {true,false}), (TooHigh, {true,false}), (TooLow, {true,false}), (MC, {Operating,Off}), (PumpOn, {true,false}), (PumpFail, {true})}. $\texttt{gen}(5)$ (node 5 is a predecessor of 50) is (PumpFail, {true,false}).

$$\mathrm{triggs} = \mathrm{events}(\mathrm{PumpFail},\ \textstyle\bigcup_{\forall p \in \{2,15,22,32\}} \texttt{out}(p),\ \texttt{gen}(5) \sqcap \texttt{when})$$

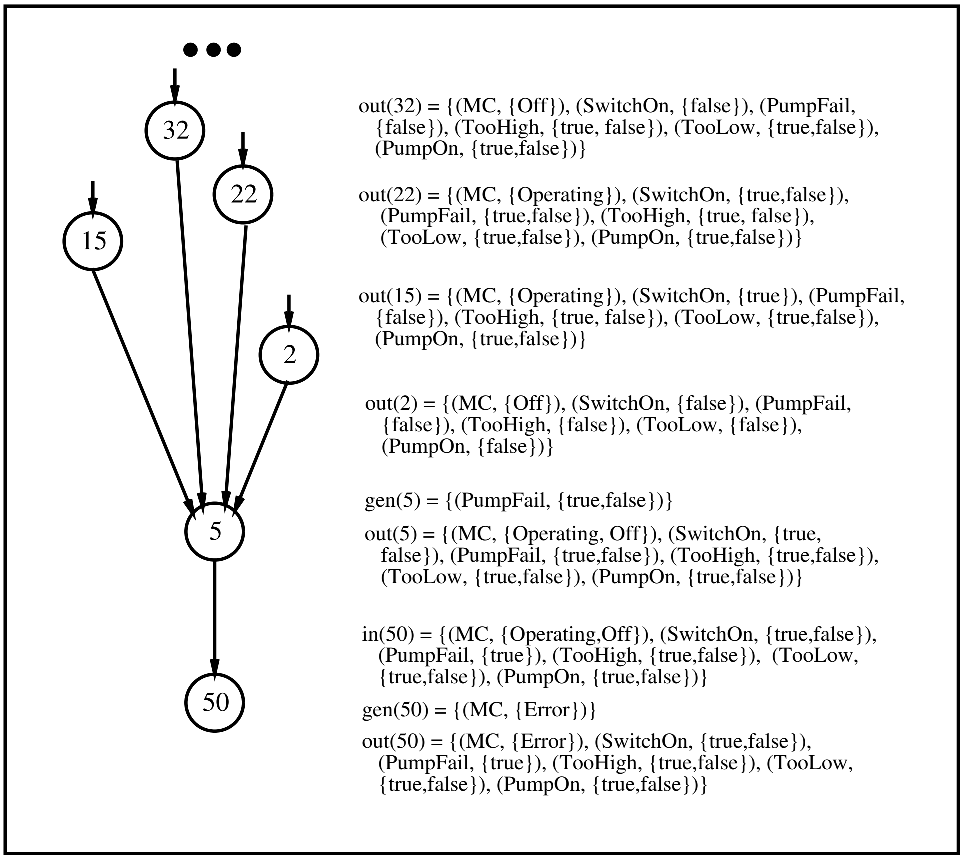


Figure 4.4: Calculating transitions for SWLMS.

Since PumpFail is {false} in nodes 2, 15 and 32, and {true,false} in node 22,

$$\bigcup_{\forall p \in \{2,15,22,32\}} \texttt{out}(p) = \{(\text{PumpFail, } \{\text{true,false}\})\}.$$

$\texttt{gen}(5) \sqcap \texttt{when} = \{(\text{PumpFail, } \{\text{true}\})\}$, which indicates that the designer assumed that PumpFail is true in node 50. The events function is called for PumpFail and yields @T(PumpFail) $\vee$ none(PumpFail). The second disjunct corresponds to paths on which PumpFail becomes true on line 22 and is again set to true on line 5. The Triggering condition has two disjuncts, @T(PumpFail) and NONE. For the event triggered by @T(PumpFail), we replace PumpFail's and MC's values with {} in $\texttt{when}[k]$ (since MC is in $\texttt{gen}(50)$). On the first iteration of the loop, `to` is (MC=Error) and `from` is

I_1: Transition from MC=Off to MC=Error:
@T(PumpFail) WHEN [{(SwitchOn, {true,false}), (TooHigh, {true,false}),
(TooLow, {true,false}), (PumpOn, {true,false}),
(PumpFail, {}), (MC, {})}]

I_2: Transition from MC=Off to MC=Error:
NONE WHEN [{(SwitchOn, {true,false}), (TooHigh, {true,false}),
(TooLow, {true,false}), (PumpOn, {true,false}),
(PumpFail, {true}), (MC, {})}]

I_3: Transition from MC=Operating to MC=Error:
@T(PumpFail) WHEN [{(SwitchOn, {true,false}), (TooHigh, {true,false}),
(TooLow, {true,false}), (PumpOn, {true,false})
(PumpFail, {}), (MC, {})}]

I_4: Transition from MC=Off to MC=Error:
NONE WHEN [{(SwitchOn, {true,false}), (TooHigh, {true,false}),
(TooLow, {true,false}), (PumpOn, {true,false}),
(PumpFail, {true}), (MC, {})}]

Figure 4.5: Transitions discovered for node 50.

(MC=Off); on the next iteration, `from` becomes (MC=Operating). The result is four transitions, I_1-I_4, for each combination of triggers and starting values of MC. These transitions are shown in Figure 4.5.

4.5 Environmental Assumptions and Events

Environmental assumptions in an SCR requirements document may make event computation more precise. Some of the relations in the semantics of these assumptions involve changes of values of variables. We use these relations in computation of Triggering and When conditions. We use a function add(event) to add events to Triggering conditions, and a function repl(var, value) to change values of variables in When conditions. repl(r_j, μ) flags an error if $\mu \not\subseteq v(r_j)$ in the When condition, i.e., if the new value is not part of the set of values of r_j in the current state. Otherwise, it sets the value of r_j to μ.

equivalence$(a = b)$

Our algorithm for computing transitions and our processing of environmental assumptions presented in previous chapters ensure that a and b already have the same values in all Triggering and When conditions.

implication $(a\text{–>}\ b)$

Refinement of events for various Triggering conditions are shown in Table 4.2. Consider the case in which the Triggering condition contains @T(b) $\wedge$ @F(b) but no clauses involving a. Some of the relations for this environmental assumption are:

$$@T(b) \rightarrow (\neg a\ \vee\ @T(a))$$
$$@F(b) \rightarrow (\neg a\ \vee\ @F(a))$$

Since neither @T(a) nor @F(a) are part of the Triggering condition, then the value of a in the When condition has to be {false}, which we indicate by repl(a, {false}).

strict implication $(a\text{–>>}\ b)$

Strict implication indicates that a and b cannot change values at the same time. However, a and b may both be present in the Triggering condition. To handle this case, we produce separate events with a and b in their respective Triggering conditions. Table

$a\backslash b$	none	@T(b)	@F(b)
none	no change	repl(a, {false})	repl(a, {false})
@T(a)	repl(b, {true})	no change	no change
@F(a)	repl(b, {true})	no change	no change

Table 4.2: Refinement of events for a–> b.

4.3 shows the refinement of When conditions for these events. Suppose the computed event is

$$@\mathrm{F}(a) \wedge @\mathrm{T}(b) \text{ WHEN } [\{(a, \{\text{true, false}\}), (b, \{\text{true,false}\}), (c{=}\{\text{true}\})\}],$$

and a–>> b. The entry for @F(a) $\wedge$ @T(b) in Table 4.3 (row 3, column 2) is repl(b, {true}) $\vee$ repl(a, {false}). The disjuncted entry means that we have to produce two different events, one with @F(a) and the value {b, true} in the When condition, and the other with @T(b) and the value {a, false} in the When condition. As usual, variables that are part of Triggering conditions are removed from the corresponding When conditions. The resulting events are

$$\begin{array}{ll} @\mathrm{F}(a) & \text{WHEN } [\{(b, \{\text{true}\}), (c, \{\text{true}\})\}] \\ @\mathrm{T}(b) & \text{WHEN } [\{(a, \{\text{false}\}), (c, \{\text{true}\})\}] \end{array}$$

timeline ($a < b$)

Timeline, like strict implication, means that some events cannot happen simultaneously. Again, the approach is to produce multiple events. Refinement of events for timeline is shown in Table 4.4.

$a \backslash b$	none	@T(b)	@F(b)
none	no change	repl(a, {false})	repl(a, {false})
@T(a)	repl(b, {true})	repl(b, {true}) $\vee$ repl(a, {false})	repl(b, {true}) $\vee$ repl(a, {false})
@F(a)	repl(b, {true})	repl(b, {true}) $\vee$ repl(a, {false})	repl(b, {true}) $\vee$ repl(a, {false})

Table 4.3: Refinement of events for a—>> b.

$a \backslash b$	none	@T(b)	@F(b)
none	no change	repl(a, {true})	add(@F(a))
@T(a)	repl(b, {false})	repl(b, {false}) $\vee$ repl(a, {true})	repl(b, {false}) $\vee$ add(@F(a))
@F(a)	repl(b, {false})	repl(b, {false}) $\vee$ repl(a, {true})	no change

Table 4.4: Refinement of events for $a < b$.

enumeration (a_1 | a_2 | ... | a_k)

This assumption does not result in any additional refinement of events.

4.6 Checking Automatically-Generated Properties

Once the FSM has been created, ALT, OLT and ENV properties are checked in a single traversal of the FSM.

4.6.1 Verification of OLT Properties

OLT properties have the general form $P_i = \forall n,\ (n \models (r_i = v_{i,new})) \rightarrow (\forall p \in \text{pred}(n),$ $(p, n) \models (r_i = v_{i,new}) \vee \bigvee_j (p, n) \models ((r_i = v_{i,j,old}) \wedge \texttt{trcond}_{i,j} \wedge \texttt{whcond}_{i,j}))$, where $v_{i,new}$ and $v_{i,old}$ are the new and the old values, respectively, for the modeclass or controlled variable r_i. $\texttt{trcond}_{i,j}$ and $\texttt{whcond}_{i,j}$ are conjuncts representing the Triggering and the When conditions of the jth row of the table entry corresponding to a change of r_j's value from $v_{i,old}$ to $v_{i,new}$. Since OLT properties are to be verified pessimistically, we want to ensure that if an OLT property is violated in the design, our analysis catches the violation. OLT violations are reported as soon as they are discovered. Since we compute a number of transitions for a single Update annotation, we can report a number of OLT violations for a given line in the design, as outlined by the algorithm in Figure 4.6.

We take advantage of the fact that all properties have been generated from SCR tables, and thus $\texttt{trcond}_{i,j}$ consists of a conjunction of one or more simple Triggering conditions (e.g., @T(a)). Our algorithm to compute transitions also results in a conjunction of simple Triggering conditions. To check that $\texttt{trigger} \rightarrow \texttt{trcond}_{i,j}$, we check that each conjunct in $\texttt{trcond}_{i,j}$ is present in $\texttt{trigger}$.

Before checking that $\texttt{when} \rightarrow \texttt{whcond}_{i,j}$, we need to represent the $\texttt{whcond}_{i,j}$ as a set of variable-value pairs. For example, PumpOn is treated as (PumpOn, {true}) and $\neg$(MC = Operating) means that MC can be either Off or Error and is treated as (MC, {Off, Error}). If a variable $r_j \in R$ is not part of $\texttt{whcond}_{i,j}$, then it was specified as a "don't care" condition in the tables, or not mentioned in the tables at all. We will refer to such variables as "don't care". The value for this variable is considered to be

Algorithm 3

Inputs: A set $\{P_i\}$ of OLT properties, where
$P_i = \forall n,\ (n \models (r_i = v_{i,new})) \rightarrow (\forall p \in \text{pred}(n), (p,n) \models (r_i = v_{i,new}) \vee$
$\bigvee_j (p,n) \models ((r_i = v_{i,j,old}) \wedge \texttt{trcond}_{i,j} \wedge \texttt{whcond}_{i,j})),$
Node n in FSM

Outputs: Error messages indicating violations of P_i's at n.

Algorithm:

Compute a set of transitions for node n.
For each transition (p, `to`, `from`, `trigger`, `when`) s.t. `to` $\neq$ `from`
    If `trigger` is equal to NONE
        Report error "no triggering conditions"
    Else {
        For each property P_i s.t. `to` $= (r_i = v_{i,new})\{$
            If no disjuct of P_i satisfies the following conditions:
                `from` $= (r_i = v_{i,j,old})$ AND
                `trigger` $\rightarrow$ $\texttt{trcond}_{i,j}$ AND
                `when` $\rightarrow$ $\texttt{whcond}_{i,j}$
            Then report a violation of P_i in node n.
        }
    }

Figure 4.6: Algorithm for verifying OLT properties.

---

a set of all of its attainable values, i.e., it is treated as $(r_j, T(r_j))$. For example, if a boolean variable TooHigh is a "don't care" for some transition, then the corresponding $\texttt{when}_{i,j}$ set contains (TooHigh, {true,false}). Once $\texttt{whcond}_{i,j}$ is represented as a set of variable-value pairs, the checking reduces to showing that

$$\texttt{when} \sqsubseteq \texttt{whcond}_{i,j}$$

`Theorem 1.` *Algorithm 3, executed at each node n of the FSM, checks all OLT properties pessimistically, i.e. if our analysis reports no violations of an OLT property, then it holds in the design.*

`Proof.` We are to prove that 1) all OLT properties are checked by our algorithm;

and 2) the verification is pessimistic.

1) One OLT property is generated for each different value that can be bound to a modeclass or controlled variable, and each formula is quantified over all the states of the FSM. Algorithm 3 is invoked only for nodes corresponding to Update annotations for modeclasses or controlled variables. We need to argue that OLT properties are properly checked at all nodes.

For nodes corresponding to Initial annotations, the right sides of the implications of all the OLT properties are vacuously true since the start node has no predecessors.

Nodes corresponding to Update annotations of monitored variables, do not generate changes to modeclass or controlled variables. Thus, if these variables have particular values in these nodes, they have the same values on all edges leading to them, e.g. $n \models (r_i = v_{i,new}) \rightarrow \forall p \in \text{pred}(n), (p, n) \models (r_i = v_{i,new})$ for each property P_i.

For nodes corresponding to Update annotations of modeclass or controlled variables, Algorithm 3 computes the maximal set of state transitions for each predecessor, and iterates over each OLT formula for each transition.

2) For a given OLT property P_i, if our algorithm did not find a violation of P_i, then for each state n reachable from the starting state s_0 and for each transition (p, `to`, `from`, `trigger`, `when`) where $p \in \text{pred}(n)$, there can be three possible cases. In the first case,

$$\texttt{to} \neq (r_j = v_{i,new}) \tag{4.1}$$

in which case P_i holds vacuously in n. In the second case, there was no transition generated for $(r_j = v_{i,new})$ since `to` and `from` were the same, in which case

$$\begin{aligned}((\texttt{to} = (r_i = v_{i,new})) \wedge (\texttt{from} = (r_i = v_{i,new}))) &\Rightarrow \\ ((n \models (r_i = v_{i,new})) \wedge (p \models (r_i = v_{i,new}))) &\Rightarrow \\ (p, n) \models (r_i = v_{i,new}) & \qquad (4.2)\end{aligned}$$

In the third case,

$$\texttt{from} = (r_i = v_{i,old}) \qquad (4.3)$$

$$\texttt{trigger} \rightarrow \texttt{trcond}_{i,j} \qquad (4.4)$$

$$\texttt{when} \rightarrow \texttt{whcond}_{i,j} \qquad (4.5)$$

A transition (p, `to`, `from`, `trigger`, `when`), computed for a node n ($p \in \mathrm{pred}(n)$), indicates that

$$\begin{aligned}\exists n, \exists p \in \mathrm{pred}(n), ((p, n) \models \texttt{from}) \wedge ((p, n) \models \texttt{when}) \wedge ((p, n) \models \texttt{trigger}) &\Rightarrow \\ (p, n) \models (\texttt{from} \wedge \texttt{when} \wedge \texttt{trigger}) & \qquad (4.6)\end{aligned}$$

From 4.6 and 4.3-4.5,

$$\exists n, \exists p \in \mathrm{pred}(n), (p, n) \models ((r_i = v_{i,old}) \wedge \texttt{whcond}_{i,j} \wedge \texttt{trcond}_{i,j}) \qquad (4.7)$$

From 4.1, 4.2 and 4.7 we get that

$$\begin{aligned}\exists n, \exists p \in \mathrm{pred}(n), (n \models (r_i = v_{i,new})) \rightarrow (p, n) \models (r_i = v_{i,new}) \vee \\ (p, n) \models ((r_i = v_{i,old}) \wedge \texttt{whcond}_{i,j} \wedge \texttt{trcond}_{i,j})) \qquad (4.8)\end{aligned}$$

Our transitions are computed in a way that all predecessors of a node are covered. From that and 4.8, we get

$$\begin{aligned}\exists n, (n \models (r_i = v_{i,new})) \rightarrow (\forall p \in \mathrm{pred}(n), (p, n) \models (r_i = v_{i,new}) \vee \\ \bigvee_j ((p, n) \models ((r_i = v_{i,old}) \wedge \texttt{whcond}_{i,j} \wedge \texttt{trcond}_{i,j}))) \qquad (4.9)\end{aligned}$$

Our algorithm is applied to each node n reachable from the starting node. From 4.9,

$$\forall n, (n \models (r_i = v_{i,new})) \rightarrow (\forall p \in \text{pred}(n), (p, n) \models (r_i = v_{i,new}) \lor \bigvee_j ((p, n) \models ((r_i = v_{i,old}) \land \texttt{whcond}_{i,j} \land \texttt{trcond}_{i,j})))$$

i.e., if the analysis does not report an error, then P_i holds in the FSM.

□

One of the OLT properties for SWLMS is

$$\begin{aligned} P_3 \quad = \quad & \forall n,\ (n \models (\text{MC=Error}) \rightarrow (\forall p \in \text{pred}(n),\ (p, n) \models (\text{MC=Error}) \ \lor \\ & (p, n) \models \text{@T(PumpFail)} \land (\text{MC=Operating}) \ \lor \\ & (p, n) \models \text{@T(PumpFail)} \land (\text{MC=Off})) \end{aligned}$$

In this property,

$\texttt{dest}_3$	is	MC=Error
$\texttt{source}_{3,1}$	is	MC=Operating
$\texttt{trcond}_{3,1}$ and $\texttt{trcond}_{3,2}$	are	@T(PumpFail)
$\texttt{whcond}_{3,1}$ and $\texttt{whcond}_{3,2}$	are	{(SwitchOn, {true,false}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {true,false})}
$\texttt{source}_{3,2}$	is	MC=Off

$\texttt{whcond}_{3,1}$ and $\texttt{whcond}_{3,2}$ were computed by converting SWLMS's variables into a set of variable-value pairs, with top value corresponding to every "don't care" variable.

For all nodes other than 50, the `to` part of transitions is not equal to MC=Error, so P_3 holds vacuously. Transitions generated for node 50 are I_1-I_4, as shown in Figure 4.5. The `from` part of I_1 = $\texttt{source}_{3,2}$, `trigger` $\rightarrow$ $\texttt{trcond}_{3,2}$ and `when` $\rightarrow$ $\texttt{whcond}_{3,2}$. So, no errors are reported for I_1. Similarly, the `from` part of I_3 = $\texttt{source}_{3,1}$, `trigger` $\rightarrow$ $\texttt{trcond}_{3,1}$ and `when` $\rightarrow$ $\texttt{whcond}_{3,1}$, and no errors are reported. `trigger`s for transitions I_2 and I_4 are NONE, indicating that no Triggering condition was found, so Analyzer reports an error message:

```
Error on line 50 of function main in modeclass MC:
   no triggering condition for transition
   from mode(s) {Operating,Off} to mode(s) {Error}
```

4.6.2 Verification of ALT Properties

All ALT properties have the general form $P_i = \exists n,\ \exists p \in \text{pred}(n),\ n \models (r_i = v_{i,new}) \wedge (p, n) \models ((r_i = v_{i,old}) \wedge \texttt{trcond}_i \wedge \texttt{whcond}_i)$, where $v_{i,old}$ and $v_{i,new}$ are the new and the old values, respectively, for the modeclass or controlled variable r_i. $\texttt{trcond}_i$ and $\texttt{whcond}_i$ are conjuncts representing the Triggering and the When conditions for this transition. ALT properties are to be verified optimistically, so we want to ensure that if the analysis finds a violation of an ALT property, this property does not hold in the design. We might not report all unimplemented transitions. Once transitions are computed for a given node, we look through the list of ALT properties and mark those which are satisfied by this transition. Any properties remaining unmarked at the end of analysis are reported as errors. An algorithm to check ALT properties is outlined in Figure 4.7. For ALT properties, we translate "don't care" conditions into empty sets. Checking the implication between the Triggering and the When conditions in requirements and designs is done the same way as for OLT properties, i.e., we check that each conjunct of `trigger` is present in $\texttt{trcond}_{i,j}$, and $\texttt{when} \sqsupseteq \texttt{whcond}_{i,j}$. Our model of SCR guarantees that there are no transitions in which the source and the destination are the same, i.e. for each P_i, $v_{i,new} \neq v_{i,old}$.

`Theorem 2.` *Algorithm 4 checks all ALT properties optimistically, i.e. if our analysis reports a violation of an ALT property, then it does not hold in the design.*

`Proof.`

Let an ALT property P_i be given. If our analysis found that P_i is violated, then

$$\forall n\ \forall p \in \text{pred}(n),\ n \not\models \texttt{to}\ \vee\ (p, n) \not\models (\texttt{from} \wedge \texttt{trigger} \wedge \texttt{when}) \tag{4.10}$$

Algorithm 4

Inputs: A set $\{P_i\}$ of ALT properties, where
$P_i = \exists n,\ \exists p \in \mathrm{pred}(n),\ n \models (r_i = v_{i,new}) \wedge$
$(p, n) \models ((r_i = v_{i,old}) \wedge \mathtt{trcond}_i \wedge \mathtt{whcond}_i)$
Finite state machine

Outputs: Error messages indicating violations of P_is at n.

Algorithm:

Unmark all ALT properties
For each node n reachable from s_0 in depth-first order
  Compute a set of transitions for n.
  For each transition (p, `to`, `from`, `trigger`, `when`)
    If there is an unmarked property P_i s.t.
      $(r_i = v_{i,new}) =$ `to` AND
      $(r_i = v_{i,old}) =$ `from` AND
      $\mathtt{trcond}_i \rightarrow$ `trigger` AND
      $\mathtt{whcond}_i \rightarrow$ `when`
    Then mark P_i
Report all unmarked ALT properties

Figure 4.7: Algorithm for verifying ALT properties.

4.10 means that one of the following is true:

$$(\forall n\ n \not\models \mathtt{to})\ \vee \tag{4.11}$$

$$(\forall n\ \forall p \in \mathrm{pred}(n),\ p \not\models \mathtt{from})\ \vee \tag{4.12}$$

$$(\forall n\ \forall p \in \mathrm{pred}(n),\ (p, n) \not\models \mathtt{trigger})\ \vee \tag{4.13}$$

$$(\forall n\ \forall p \in \mathrm{pred}(n),\ (p, n) \not\models \mathtt{when}) \tag{4.14}$$

If 4.11 is true, then, considering that we would have marked P_i if $(r_i = v_{i,new}) = \mathtt{to}$,

$$\forall n\ n \not\models (r_i = v_{i,new}) \tag{4.15}$$

If 4.12 is true and $\mathtt{to} \neq \mathtt{from}$, then, considering that we would have marked P_i if $(r_i = v_{i,old}) = \mathtt{from}$,

$$\begin{aligned} &\forall n\ \forall p \in \mathrm{pred}(n),\ p \not\models \mathtt{from} \Rightarrow \\ &\forall n\ \forall p \in \mathrm{pred}(n),\ (p, n) \not\models \mathtt{from} \end{aligned} \tag{4.16}$$

If 4.13 is true, then, considering that we would have marked P_i if $\texttt{trcond}_i \rightarrow \texttt{trigger}$

$$\forall n\ \forall p \in \mathrm{pred}(n),\ (p, n) \not\models \texttt{trcond}_i \tag{4.17}$$

Finally, if 4.14 is true, then, considering that we would have marked P_i if $\texttt{whcond}_i \rightarrow$ $\texttt{when}$,

$$\forall n\ \forall p \in \mathrm{pred}(n),\ (p, n) \not\models \texttt{trcond}_i \tag{4.18}$$

Since one of 4.10-4.13 has to be true (or else the algorithm would have marked P_i), then

$$\begin{aligned}(\forall n\ n \not\models (r_i = v_{i,new}))\ \vee \\ (\forall n\ \forall p \in \mathrm{pred}(n),\ (p, n) \not\models (r_i = v_{i,old}))\ \vee \\ (\forall n\ \forall p \in \mathrm{pred}(n),\ (p, n) \not\models \texttt{trcond}_i)\ \vee \\ (\forall n\ \forall p \in \mathrm{pred}(n),\ (p, n) \not\models \texttt{whcond}_i)\end{aligned} \tag{4.19}$$

4.19 means that P_i does not hold.

$\square$

The following are some of the ALT properties for SWLMS:

$$\begin{aligned} P_7 &= \exists n,\ \exists p \in \mathrm{pred}(n),\ n \models (\mathrm{MC{=}Error}) \wedge (p, n) \models ((\mathrm{MC{=}Off}) \wedge @\mathrm{T}(\mathrm{PumpFail})) \\ P_9 &= \exists n,\ \exists p \in \mathrm{pred}(n),\ n \models (\mathrm{MC{=}Error}) \wedge \\ &\quad (p, n) \models ((\mathrm{MC{=}Operating}) \wedge @\mathrm{T}(\mathrm{PumpFail}))\end{aligned}$$

In these properties,

`dest`$_7$ and `dest`$_9$	are	MC=Error
`source`$_7$	is	MC=Off
`trcond`$_7$ and `trcond`$_9$	are	@T(PumpFail)
`source`$_9$	is	MC=Operating
`whcond`$_7$ and `whcond`$_9$	are	{(SwitchOn, {}), (TooHigh, {}), (TooLow, {}), (PumpOn, {})}

`whcond`$_7$ and `whcond`$_9$ were computed by converting SWLMS's variables into a set of variable-value pairs, with {} value corresponding to every "don't care" variable.

Transitions generated for node 50, which is reachable from the start state, enable the algorithm to mark P_7 and P_9 as satisfied. The `from` part of I_1 is MC=Off, the `to` part is MC=Error, `trcond`$_7$ $\rightarrow$ `trigger` and `whcond`$_7$ $\rightarrow$ `when`. So, P_7 is satisfied by I_1. Similarly, P_9 is satisfied by I_3.

4.7 Verifying User-Specified Safety Properties

Users can specify five types of properties: reach, smi, wmi, strcause and cause. Each of these properties is verified in a separate traversal of the FSM. reach properties are verified optimistically, whereas the remaining properties are verified pessimistically. We use `out`(n) to check that a formula holds in state n.

- reach(P): Analyzer searches the state space for a state where P holds, and if one is found, the entire property is considered to hold.

- smi(Mc=Md, P): These properties are equivalent to $\forall n$, $(n \models (\text{Mc=Md})) \rightarrow (n \models \text{P})$. During their verification, Analyzer takes all states where (Mc=Md) holds, and determines if P holds in each of these states. The process continues throughout the FSM.

- wmi(Mc=Md, P): These properties are equivalent to $\forall n,\ (n \models (\text{Mc=Md})\ \wedge (\forall p \in \text{succ}(n), (p \models (\text{Mc=Md})))) \rightarrow (n \models \text{P})$. During their verification, Analyzer looks for the nodes such that Mc=Md holds in the node and in at least one of its successors, and checks that P holds in this node.

- cause(P_1, P_2): Analyzer starts with all states where P_1 holds and checks if P_2 holds there; if not, it looks at all successor states and reports an error if P_2 does not hold in one of these.

- strcause(P_1, P_2): This property is verified like cause(P_1, P_2) except that only successor states are examined.

4.8 Example - Verification of SWLMS

We ran Analyzer on a design of the SWLMS which contains several errors. The design is shown in Figure 4.8. Analyzer reported the following errors in the design:

- The transition to mode Off on line 32 is triggered by changing the water level (line 29) rather than turning off the switch. This results in one OLT violation because the transition is illegal, and one ALT violation because a requirements transition is not implemented.

- The pump is not turned off when the pump failure is detected (line 48). This results in two ALT violations, for transitions from modes Off and Operating.

- Changes of value of PumpOn on line 38 might be caused by events other than those specified in the requirements. This results in one OLT violation.

```
{ ...
  @@ Initial MC=Off & SwitchOn=false & PumpFail=false &
     TooHigh=false & TooLow=false & PumpOn=false;
  while(1) {
    READ_DEVICE();
    @@ Update PumpFail=Top;
    if (PUMP_FAIL()) {
      @@ Assert PumpFail=true;
      break;
    }
    @@ Assert PumpFail=false; /* assume no device failures */
    READ_SWITCH();            /* read switch monitor */
    @@ Update SwitchOn=Top;
    if (SWITCH_ON() && IN_OFF(System)) {
      @@ Assert SwitchOn=true & MC=Off;
      @@ Update MC=Operating;
    }
    else if (IN_OPERATING(System)) {
      @@ Assert MC=Operating;
      while(1) {
        READ_DEVICE();
        @@ Update PumpFail=Top;
        if (PUMP_FAIL()) {
          @@ Assert PumpFail=true;
          break;
        }
        @@ Assert PumpFail=false;
        GET_WATER_LEVEL (&Water); /* compute water level */
        @@ Update TooHigh=Top & TooLow=Top;
        if (!SWITCH_ON()) {
          @@ Assert SwitchOn=false;
          @@ Update MC=Off;
          break;
        }
        @@ Assert SwitchOn=true;
        if (IS_HIGH(Water) || IS_LOW(Water)) {
          @@ Assert TooHigh=true | TooLow=true;
          @@ Update PumpOn=true;
        }
        else {
          @@ Assert TooHigh=false & TooLow=false;
          @@ Update PumpOn=false;
        }
      }
    }
  }
  @@ Assert PumpFail=true;
  @@ Update MC=Error;
}
```

Figure 4.8: SWLMS design containing errors.

- There are paths through the design that do not produce changes of values of monitored variables intended to trigger mode transitions and changes of values of controlled variables. This results in five "no events found" violations.

- The pump is not turned on as soon as the water level becomes too high (lines 29-32). This results in a violation of the user-defined property

  cause[((MC=Operating) & TooHigh), PumpOn]

Messages produced by the tool are shown in Figure 4.9.

4.9 Summary

We have defined a notion of consistency between SCR-style requirements and a detailed design. We have also presented a technique to check that this notion of consistency is satisfied, implemented in a tool called Analyzer. Analyzer creates a finite-state abstraction of a detailed design and checks it against a set of properties automatically generated from the requirements or specified by the user. Figure 4.10 summarizes the steps of the analysis.

We envision Analyzer being used in a software development process in which SCR specifications are first written and checked for consistency and completeness. Then, a design is developed using our PDL. A detailed design is automatically verified for consistency with the requirements, also ensuring that global requirements properties are preserved. Afterwards, a real implementation is written around the PDL statements. Consistency checking between code and design statements is assured through code reviews. This process gives a developer some assurance that the code implements the system behavior specified in the requirements. Finally, the implementation is thoroughly tested.

```
                    ANALYZER (Version 3.02)

Error on line 15 of function main in modeclass MC:
   no triggering condition for transition
   from mode(s) {Off} to mode(s) {Operating}

Error on line 32 of function main in modeclass MC:
   no triggering condition for transition
   from mode(s) {Operating} to mode(s) {Off}

Error on line 32 of function main in modeclass MC:
   illegal transition from mode(s) {Operating} to mode(s) {Off}.
Event is (@T(TooHigh) & @T(TooLow)) or (@T(TooHigh) & @F(TooLow))
   or (@F(TooHigh) & @T(TooLow)) or (@F(TooHigh) & @F(TooLow))
   WHEN [{(PumpOn, {true,false}), (SwitchOn, {false}),
        (PumpFail, {false})}]
Event is @T(TooHigh) or @F(TooHigh) WHEN [{(PumpOn, {true,false}),
   (SwitchOn, {false}), (TooLow, {true,false}), (PumpFail, {false})}]
Event is @T(TooLow) or @F(TooLow) WHEN [{(PumpOn, {true,false}),
   (SwitchOn, {false}), (TooHigh, {true,false}), (PumpFail, {false})}]

Error on line 38 of function main in variable PumpOn:
   no triggering condition for transition from FALSE to TRUE

Error on line 38 of function main in variable PumpOn:
   illegal transition from FALSE to TRUE
Event is @F(TooHigh) WHEN [{(MC, {Off,Operating}), (SwitchOn, {true}),
   (TooLow, {true,false}), (PumpFail, {false})}]
Event is @F(TooLow) WHEN [{(MC, {Off,Operating}), (SwitchOn, {true}),
   (TooHigh, {true,false}), (PumpFail, {false})}]

Error on line 42 of function main in variable PumpOn:
   no triggering condition for transition from TRUE to FALSE

Error on line 48 of function main in modeclass MC:
   no triggering condition for transition
   from mode(s) {Operating,Off} to mode(s) {Error}
```

Figure 4.9: Output of analysis of SWLMS.

```
                 *** MISSING TRANSITIONS ***

 FROM                    EVENT                                TO
-----------------------------------------------------------------
 Operating      @F(SwitchOn) WHEN [~PumpFail]                  Off

 VARIABLE                EVENT                                VALUE
------------------------------------------------------------------
 PumpOn         @T(PumpFail) WHEN [MC=Off]                    false
 PumpOn         @T(PumpFail) WHEN [MC=Operating]              false

             *** VERIFICATION OF PROPERTIES ***

 Violation of property Cause [(MC=Operating & TooHigh), PumpOn]
 on line 29 by a successor on line 32.
 The state is {(MC, {Operating}), (PumpOn, {true,false}),
 (SwitchOn, {true,false}), (PumpFail, {false}), (TooHigh,
 {true,false}), (TooLow, {true,false})}
 The successor state is {(MC, {Off}), (PumpOn, {true,false}),
 (SwitchOn, {false}), (PumpFail, {false}), (TooHigh,
 {true,false}), (TooLow, {true,false})}

               *** VIOLATION SUMMARY ***

OLT properties for modeclasses:          1 time (1 location, 9 events)
OLT properties for controlled variables: 1 time (1 location, 2 events)
OLT properties (no events found):        5 times
ALT properties for modeclasses:          1 time
ALT properties for controlled variables: 2 times
User-defined properties:                 1 time
```

Figure 4.9: Output of analysis of SWLMS (cont'd).

Algorithm 5

Inputs: A set of SCR requirements $\mathcal{R}$,
A program design (PD).
Outputs: Messages reporting violations of our definition of consistency between $\mathcal{R}$ and PD.
Algorithm:

1. Generate $\mathcal{P}$ - a set of properties to ensure that PD is consistent with $\mathcal{R}$. $\mathcal{P}$ consists of:
   All transitions are implemented (ALT properties)
   No illegal transitions are implemented (OLT properties)
   All FSM nodes are reachable (REACH),
   All user-defined properties hold (USER), and
   Environmental assumptions are preserved (ENV).
2. Build DFG:
   Compute information generated at each node and propagate it through the control-flow graph until a fixed point is reached.
   Check that ENV properties hold.
3. Abstract DFG to FSM:
   For each node containing an Update or an Initial annotation, check the REACH property. If violated, report violation and mark the node unreachable. Connect all reachable nodes.
4. Check that FSM $\models \mathcal{P}$:
   Expand When conditions for OLT and ALT properties.
   For each reachable transition (p, `to`, `from`, `trigger`, `when`) {
   Verify relevant OLT properties.
   Mark corresponding ALT properties as implemented.
   Report OLT property violations.
   }
   Report unmarked ALT properties, if any.
5. Verify USER properties of the system.

Figure 4.10: Summary of the analysis.

Chapter 5

Case-Study

To demonstrate our analysis technique (described in Chapters 2-4) on a more realistic application, we conducted a case study of a Water-Level Monitoring System (WLMS) which had been specified using SCR requirements and subsequently implemented[van90]. To create the design, we reversed engineered an existing implementation of WLMS in order to determine what, if any, errors might be detected with our methods.

5.1 The Application

A Water-Level Monitoring System (WLMS) is a safety monitoring device, which serves as a component of a steam generator application. It ensures that the water level is within a specified range while the steam generator is in operation. Inside this allowable water-level range, there is a specified hysteresis water-level range. The area inside the allowable water-level range but outside the hysteresis range is used as a buffer to keep the generator from toggling on and off when the water level is near the limits of its allowable range. The system also raises visual and audio alarms and shuts off its pump when the level is out of range or when the monitoring system fails. In addition, there are two buttons on the system control console: the SelfTest button permits the operator to test the system, and the Reset button permits the operator to return the monitoring system (and the steam generator) to normal operation, provided that the water level is inside the hysteresis range. A complete description of this system can be found in [van90]. WLMS has two mode classes, Normal and Failure. Modeclass Normal is comprised of four modes:

Operating	The system is running properly.
Shutdown	The water level is out of range and the system will be shutdown unless conditions change.
Standby	The system is waiting for the operator to push a button to select test or operating mode.
Test	The system is not operating, but controlled variables are being checked.

Modeclass Failure is comprised of three modes:

AllOK	No device failures.
BadLevDev	The water level cannot be measured.
HardFail	Unrecoverable failure.

The system starts in mode Standby of modeclass Normal and mode AllOK of modeclass Failure. A mode transition table for modeclass Normal is shown in Table 5.1. Conditions on monitored variables which appear in the table are defined as follows:

WithinLimits	LowLimit $<$ WaterLevel $<$ HighLimit
InsideHysR	(LowLimit+0.5cm) $<$ WaterLevel $<$ (HighLimit-0.5cm)
SlfTstPressed	SelfTest button is being pressed
SlfTstPressed500	SelfTest button pressed constantly for $>$ 500ms
ResetPressed	Reset button is being pressed
ResetPressed3000	Reset button pressed constantly for $>$ 3s
InTest0	The system entered mode Test
InTest2000	The system has been in Test for $>$ 2s
InTest4000	The system has been in Test for $>$ 4s
InTest14000	The system has been in Test for $>$ 14s
ShutdownLockTime200	System has been in Shutdown mode for $>$ 200ms

Environmental assumptions which appear in Table 5.1 have been deduced from descriptions of the conditions in the requirements.

Controlled variables are set to trigger alarms and to display the water level to the operator. Table 5.2 shows the settings for the controlled variable LowWindowOn which represents the annunciation window labeled “Water Level Low”. When LowWindowOn is true, the annunciator displays a value. This happens when the water level falls below its limits (i.e., LevelLow) or when the water level cannot be measured (BadLevDev) and a certain amount of time has passed (FlashOff500).

Current Mode	Inside HysR	Within Limits	SlfTst Pressed	SlfTst Pressed 500	In Test 14000	Reset Pressed 3000	Shutdown LockTime 200	**New Mode**
Standby	t	–	–	–	–	@T	–	Operating
	–	–	–	@T	–	–	–	Test
Operating	–	@F	f	–	–	–	–	Shutdown
	–	–	–	@T	–	–	–	Test
Shutdown	@T	–	f	–	–	–	f	Operating
	–	–	f	–	–	–	@T	Standby
	–	–	–	@T	–	–	–	Test
Test	–	–	–	–	@T	–	–	Standby

Initial: Standby ($\neg$SlfTstPressed500 $\wedge$ $\neg$ResetPressed3000 $\wedge$ $\neg$InTest14000 $\wedge$ $\neg$ShutdownLockTime200)

Assumptions: InsideHysR -> WithinLimits
SlfTstPressed < SlfTstPressed500
ResetPressed < ResetPressed3000
LDTestVal | LD0
InTest0 < InTest2000 < InTest4000 < InTest14000

Table 5.1: Mode transition table for mode class Normal.

Mode	Triggering Event	
Operating $\wedge$ AllOK	@T(LevelLow)	@T(Inmode) WHEN [$\neg$LevelLow]
Shutdown $\wedge$ AllOK	@T(LevelLow)	
Test $\wedge$ AllOK	@T(InTest2000)	@T(InTest4000)
BadLevDev	@F(FlashOff500)	@T(FlashOff500)
LowWindowOn =	true	false

Initial: true

Table 5.2: Event table for controlled variable LowWindowOn.

The requirements included four user-defined safety properties, identical to those used in [AG93]:

$$\begin{aligned} U_1 &= \text{cause(SlfTstPressed500, Normal(Test))} \\ U_2 &= \text{wmi(Normal=Standby, } \neg\text{SlfTstPressed500)} \\ U_3 &= \text{wmi(Normal=Operating, (}\neg\text{SlfTstPressed500} \wedge \\ &\quad \text{(WithinLimits} \vee \text{SlfTstPressed)))} \\ U_4 &= \text{wmi(Normal=Shutdown, (}\neg\text{SlfTstPressed500} \wedge \\ &\quad \text{((}\neg\text{InsideHysR} \wedge \neg\text{ShutdownLockTime200)} \vee \neg\text{SlfTstPressed)))} \end{aligned}$$

If the SelfTest button has been pressed for 500ms or more, the system is either in mode Test or will be in mode Test after its next transition. When the system is in mode Standby, the SelfTest button has not been pressed for 500ms. If the system is in mode Operating, then either the water level is WithinLimits or the SelfTest button has not been pressed long enough to cause a transition to mode Test. If the system is in mode Shutdown, then either the water level is outside the hysteresis water-level range and the system has been in mode Shutdown for less than 200 ms, or the SelfTest button has been pressed but not long enough to cause a transition to mode Test.

5.2 Creating Designs

To build a design, we reverse engineered an existing implementation of the WLMS, originally consisting of roughly 1300 lines of FORTRAN and Assembler code. To analyze the program, we translated it into C and changed the Assembler routines for manipulating the screen with a GUI written in Xlib. Then we reverse engineered the implementation, producing a detailed design. This was a manual process, which entailed determining the meaning of the code in terms of requirements, and capturing

```
while (true) {
  ...
  GET_MODE(OperatingClass, OPCM);
  WATER_LEVEL(&LVLCM);
  INLIM = ((LVLCM < (float)DPAR_HWL) && (LVLCM > (float)DPAR_LWL));
  GET_TIME(SHUTER, &SHUTTM);
  if (ERR_VAR_F(CLK_ERR) == CLK_OVERFLOW) {
    RESET_TIMER(SHUTER);
    SHUTTM = 0;
  }
  if (MODE_EQ(OPCM, Shutdown) && INLIM && (SHUTTM < DPAR_SLT)
    && !PRESSED(SlfTst)) {
    SET_MODE (OperatingClass, Operating);
  }
  ...
  GET_MODE(OperatingClass, OPCM);
  GET_TIME(SHUTER, &SHUTTM);
  if ((!MODE_EQ(OPCM, Test)) && (Pressed(SlfTst))) {
    SET_MODE(OperatingClass, Test);
    INIT_DRVS();
  }
  ...
}
```

Figure 5.1: Fragment of the WLMS implementation, translated to C.

programmer assumptions and state changes in the PDL language. A fragment of the WLMS implementation, translated to C is shown in Figure 5.1, and the same fragment with PDL constructs is shown in Figure 5.2. The application was written without the concept of events, and thus at times it was difficult to determine the exact time when an event occurred. For example, many times the program compared the shutdown timer against a threshold (SHUTTM < DPAR_SLT) or used a function to determine if the SelfTest button was pressed (Pressed (SlfTst)). In our first attempt, we tried to annotate the code so that these checks constituted events, so that comparison of shutdown timer against a threshold results in an annotation @@Update Shutdown-

```
while (true) {
  ...
  GET_MODE(OperatingClass, OPCM);
  WATER_LEVEL(&LVLCM);
  INLIM = ((LVLCM < (float)DPAR_HWL) && (LVLCM > (float)DPAR_LWL));
  @@ Update WithinLimits=Top;
  GET_TIME(SHUTER, &SHUTTM);
  @@ Update ShutdownLockTime200=Top;
  if (ERR_VAR_F(CLK_ERR) == CLK_OVERFLOW) {
    RESET_TIMER(SHUTER);
    SHUTTM = 0;
  }
  if (MODE_EQ(OPCM, Shutdown) && INLIM && (SHUTTM < DPAR_SLT)
    && !PRESSED(SlfTst)) {
    @@ Assert Normal=Shutdown & WithinLimits=true & SlfTstPressed=
       false & ShutdownLockTime200=false;
    SET_MODE (OperatingClass, Operating);
    @@ Update Normal=Operating;
  }
  ...
  GET_MODE(OperatingClass, OPCM);
  GET_TIME(SHUTER, &SHUTTM);
  @@ Update ShutdownLockTime200=Top;
  if ((!MODE_EQ(OPCM, Test)) && (Pressed(SlfTst))) {
    @@ Assert Normal=Shutdown & SlfTstPressed=false;
    SET_MODE(OperatingClass, Test);
    @@ Update Normal=Standby;
    INIT_DRVS();
  }
  ...
}
```

Figure 5.2: Fragment of the WLMS implementation with PDL constructs.

LockTime=false. However, an alternative interpretation is also possible.

In the second design, we annotated the code with Update annotations when a value of a sensor was read (in these cases the variable corresponding to the sensor was assigned the Top value) and with Assert annotations when a value of a sensor was tested. For example, reading the shutdown timer corresponds to @@Update ShutdownLock-

Time200=Top, and testing that the timer was less than a threshold value corresponded to @@Assert ShutdownLockTime200=false. Once we found a correspondence between variables on the requirements and the implementation levels, it was remarkably easy to annotate the code.

Both versions of the design were about 700 lines long. The first version contained 30 Update annotations corresponding to monitored variables, 32 Update annotations corresponding to modeclass and controlled variables, and 34 Assert annotations. The second version contained 22 Update annotations corresponding to monitored variables, 31 Update annotations corresponding to mode class and controlled variables, and 56 Assert annotations. Out of 54 functions in the original program, only eight had state changes and thus were included into the design. The implementation of WLMS together with annotations corresponding to the second version of the design is shown in Appendix C.1.

5.3 Analysis

Description	Time in seconds		
	Real	User	Sys
Design version 1, set-based approximation	13.7	10.0	1.5
Design version 2, set-based approximation	12.0	8.5	1.3
Design version 1, exact analysis	17834.5	12644.2	3082.1
Design version 2, exact analysis	14472.6	11035.3	2702.3

Table 5.3: Time of analyzing different versions of the WLMS.

We ran the analysis on a SPARC 5 with 110 MHz microSPARC II CPU and 64 megabytes of memory. The time to analyze the design of WLMS varied greatly between the set-based approximation and the exact analysis approaches, but little between the two different versions of the design. Table 5.3 shows the analysis time as measured by the UNIX **time** command. We found that it was necessary to rerun the analysis a number of times before we could get annotations right, and thus a quick response from a version of the tool implementing set-based approximation was appreciated.

Property Type	Version 1		Version 2	
	Messages	Violations	Messages	Violations
REACH property	4		10	
OLT properties for modeclasses	34	5	15	8
OLT properties for controlled variables	148	4	37	12
No events found	16		15	
ALT properties for modeclasses	10		13	
ALT properties for controlled variables	29		21	
User-defined properties	1		1	

Table 5.4: Results of analyzing the WLMS.

After we eliminated annotation errors, Analyzer reported a number of inconsistencies between the requirements and the design. Table 5.4 shows results of running the set-based approximation analysis on both versions of the design. Messages produced by Analyzer are organized in two columns: "Messages" indicates the actual number of messages, and "Violations" indicates the number of invalid mode transitions or changes of values of controlled variables. The actual output of the analysis for the second version of the design is shown in Appendix C.2. These numbers overestimate the actual errors in the design. First, some mode transitions and controlled variable value changes resulted in a number of OLT property violations. For example, eight illegal mode transitions in the second version of the design generated 15 violation messages because several illegal transitions were detected at each location. All of the mode transition problems can be attributed to four principal causes:

1. the wrong monitored variable was checked to enable mode transitions (WithinLimits rather than InsideHysR),
2. the times that the operator pressed the SelfTest and Reset buttons were not calculated or checked,
3. some events were computed several statements before mode transitions triggered by them occurred,
4. and no transitions to a mode corresponding to the complete system failure were implemented.

Most of the illegal assignments to the controlled variables occurred because the order of triggering events in the design differed from that in the requirements. Finally, even though ALT properties were checked optimistically, we found that a large portion of the specification was not implemented at all.

Each of the first three user-defined properties was verified by Analyzer. These results are not particularly interesting because all the safety properties included a negated monitored variable (SlfTstPressed500) that was never assigned the value true in the design. Thus, the first two formulae hold trivially. The third formula:

$$\text{wmi}(\text{Normal=Operating}, (\neg\text{SlfTstPressed500} \wedge (\text{WithinLimits} \vee \text{SlfTstPressed})))$$

was verified since WithinLimits was set to true before each transition to Operating, and whenever WithinLimits and SlfTstPressed were set to false, the system immediately transitioned to mode Shutdown. Analyzer reported a violation of the fourth property:

$$\text{wmi}(\text{Normal=Shutdown}, (\neg\text{SlfTstPressed500} \wedge$$
$$((\neg\text{InsideHysR} \wedge \neg\text{ShutdownLockTime200}) \vee \neg\text{SlfTstPressed}))$$

because InsideHysR received the value {true,false} as a result of its relationship to WithinLimits. This value was propagated throughout the graph, and, since wmi properties are verified pessimistically, Analyzer reported a potential fault. In fact, InsideHysR is never explicitly set to true in the code, and thus this property holds in the design.

Our experiments showed the difference between the two versions of annotated design was not significant, although the second version was much easier to produce.

5.4 Discussion

SCR requirements describe a black-box behavior of a system, and a program that just reads monitored variables and produces correct controlled variables is considered to correspond to its specification. Modes and modeclasses are logical entities allowing us to specify the conditions under which the system is to produce the required values for controlled variables. Using modeclass information in writing designs seems quite natural, and we found it encouraging that the studied implementation of the WLMS had modeclasses.

Chapter 6

Analyzing Concurrent Designs

In previous chapters we described a technique to analyze consistency between sequential designs and their requirements. The number of states before a fixed point is reached during dataflow analysis grows linearly with the number of variables in the system. Still, to ensure applicability of the approach to realistic systems, we have to develop compositional techniques. One possible way to break up the problem is suggested by our requirements notation which naturally models concurrently executing processes as separate modeclasses. If we place certain restrictions on the channels of communication between these processes, we can analyze the system compositionally. Thus, we are interested in extending our analysis methods to process designs of concurrent programs.

Most of the previous approaches to analysis of concurrent programs have generally abstracted away data values and their effects on control flow to concentrate on task interactions (e.g., [DC94, FG94, LC91, GL92, Tay83, MR91]). We perform dataflow analysis on values of requirements variables and include them in the property verification, without building a global reachability graph.

In this chapter we use a simple example to introduce new extensions to the requirements and the design languages. We also describe changes to the analysis tool which enable us to build finite-state abstractions and verify properties of concurrent designs.

6.1 Extensions to Requirements Notation

6.1.1 Expressing Communication

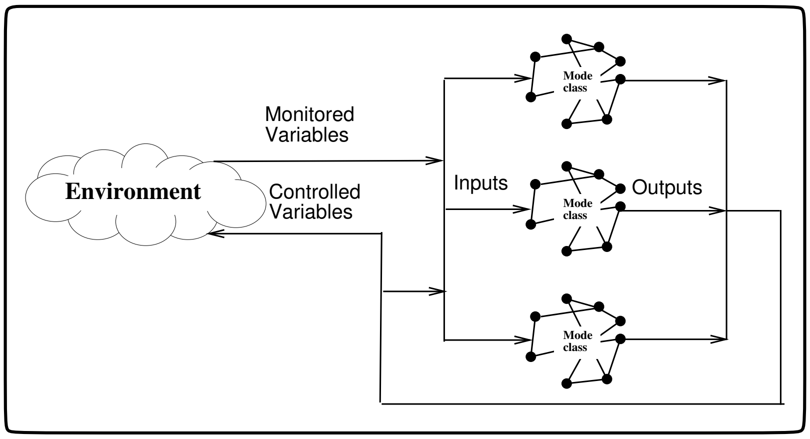


Figure 6.1: SCR requirements notation.

SCR views mode classes as independent concurrently-executing state machines. So, a natural way to break up a problem is to implement each mode class in a separate process. To be able to analyze each process in turn, we make certain restrictions to the original SCR model. Each mode class takes a set of Inputs (monitored variables) and produces Outputs (controlled variables). One mode class communicates with another by sending the values of some of its outputs to the other mode class which treats them as part of its set of inputs. Only some of the total inputs and outputs are used to communicate with the environment, as controlled or monitored variables (see Figure 6.1). The rest are called communication variables. The following restrictions are imposed on these variables:

1. a is a communication variable if a is a controlled variable for some modeclass mc_1 and monitored variable for another modeclass mc_2.

2. For each communication variable a there is only one mode class which can modify

it, i.e., a is a controlled variable for only one mode class.

3. If a is a monitored variable of modeclass mc, then it is not a controlled variable of mc and vice versa.

We use COM to represent the set of all communication variables. Let COM_{out,mc_j} and COM_{in,mc_j} be a set of communication variables used as an output or as an input to modeclass mc_j, respectively. So,

$$COM = \bigcup_{mc_j \in MC} COM_{out,mc_j}$$

Note that we do not ensure that there are no circular definitions, i.e., a partial order of dependencies between variables and mode classes in each requirements document can be constructed (as in [HLK95]). If mc_1 and mc_2 are mode classes, then we allow mc_1 to react to changes of state of mc_2, and for mc_2 to react to changes of state of mc_1.

6.1.2 Example - Mutual Exclusion System

We illustrate the SCR model using a simple example, abstracted from a Nuclear Control Rod system [JM87] to preserve communication problems. We call it a Mutual Exclusion system (MES). Assume that we want to specify a system where two identical processes have critical sections, and mutual exclusion is guaranteed by having a separate process, Manager, give out permissions to the other processes. The requirements for this system consist of three mode classes running in parallel: mode classes SubSys1 and SubSys2 describe two identical subsystems, and mode class Manager determines which subsystem has permission to enter its critical section. The modes of Manager are

Stable	Manager is waiting for requests.
Grant1	Grant is given to subsystem 1.
Grant2	Grant is given to subsystem 2.

The modes of SubSys1 are

S1Done	Subsystem is doing work in non-critical section.
S1Wait	Subsystem is waiting for a permission to enter its critical section.
S1RecGrant	Subsystem is in its critical section.

The modes of mode class SubSys2 are similar to those in SubSys1. Since the mode classes are running in parallel, the entire system MSE is

$$MSE = Manager \parallel SubSys1 \parallel SubSys2,$$

where $\parallel$ indicates a parallel composition of the processes. SCR requirements are typically used to specify sequential systems. When each mode transition table is implemented in a separate process, race conditions may occur when composing transition tables. We assume that subsystems cannot make simultaneous state changes, and this naturally gives us an interleaving model (see Section 6.2.3).

The subsystems communicate with Manager by sending it messages (either requesting or returning access permission). In the case of a request, a subsystem waits until the permission is granted. Communication variables In(SubSys1=S1Wait), In(SubSys2=S2Wait), In(SubSys1=S1Done) and In(SubSys2=S2Done) are used by Manager to determine which of the two processes is waiting to receive permission. These variables are outputs of the respective subsystems and inputs to Manager. In(Manager=Grant1) and In(Manager=Grant2) are used by the subsystems to determine which one is allowed to proceed; they are outputs of Manager and inputs to the subsystems. All these variables are state conditions.

Definition 6.1 *A state condition In(mc=md) indicates that mode class mc is in mode md. It becomes true when the system enters mode md, and false when it leaves this mode.*

State conditions are modifications of InMode() function (see Section 2.2.2).

Current Mode	Time	In(Manager=Grant1)	Done	**New Mode**
S1Done	@T	–	–	S1Wait
S1Wait	–	@T	–	S1RecGrant
S1RecGrant	–	–	@T	S1Done

Initial: S1Done ($\neg$Time $\wedge$ $\neg$Done)

Table 6.1: Mode transition table for SubSys1.

Current Mode	In(SubSys1 =S1Wait)	In(SubSys2 =S2Wait)	In(SubSys1 =S1Done)	In(SubSys2 =S2Done)	**New Mode**
Stable	@T	f	–	–	Grant1
	f	@T	–	–	Grant2
Grant1	–	–	@T	–	Stable
Grant2	–	–	–	@T	Stable

Initial: Stable (In(SubSys1=S1Done) $\wedge$ In(SubSys2=S2Done))

Table 6.2: Mode transition table for Manager.

In addition, each subsystem monitors the variable Time to determine when it is time to ask for permission, and Done to determine when critical-section processing is complete. The requirements specification for mode classes SubSys1 and Manager are shown in Tables 6.1 and 6.2, respectively. For example, Manager transits from Stable to Grant1 when SubSys1 has made a request (In(SubSys1=S1Wait) changes value from false to true) while SubSys2 is not waiting; and to Grant2 when SubSys2 has made a request while SubSys1 is not waiting. We assume that both systems cannot request access permission at the same time, so that our model is interleaving. Mode class Manager starts in mode Stable, and both subsystems start in mode Done. Communication variables of this system are:

$$
\begin{array}{lcl}
COM_{out,Manager} & = & \{\text{In(Manager=Grant1), In(Manager=Grant2)}\} \\
COM_{in,Manager} & = & \{\text{In(SubSys1=S1Wait), In(SubSys2=S2Wait),} \\
 & & \{\text{In(SubSys1=S1Done), In(SubSys2=S2Done)}\} \\
COM_{out,SubSys1} & = & \{\text{In(SubSys1=S1Wait), In(SubSys1=S1Done)}\} \\
COM_{in,SubSys1} & = & \{\text{In(Manager=Grant1)}\} \\
COM_{out,SubSys2} & = & \{\text{In(SubSys2=S2Wait), In(SubSys2=S2Done)}\} \\
COM_{in,SubSys2} & = & \{\text{In(Manager=Grant2)}\} \\
\hline
COM & = & \{\text{In(Manager=Grant1), In(Manager=Grant2),} \\
 & & \text{In(SubSys1=S1Wait), In(SubSys1=S1Done),} \\
 & & \text{In(SubSys2=S2Wait), In(SubSys2=S2Done)}\}
\end{array}
$$

The following is the summary of the safety properties we are interested in proving for this system:

$$
\begin{array}{lcl}
U_1 & = & \text{reach (Manager=Grant1)} \\
U_2 & = & \text{reach (Manager=Grant2)} \\
U_3 & = & \text{smi}(\neg(\text{(SubSys1=S1RecGrant)} \wedge \text{(SubSys2=S2RecGrant)})) \\
U_4 & = & \text{strcause(SubSys1=S1Done, Manager=Stable)} \\
U_5 & = & \text{strcause(SubSys2=S2Done, Manager=Stable)}
\end{array}
$$

The first two properties indicate that both grants are guaranteed to be given out at some point, i.e., the system can reach a state in which Manager=Grant1 holds, and a state in which Manager=Grant2 holds. U_3 is used to capture the mutual exclusion between subsystems, i.e. it is invariantly true that the two subsystems cannot be in their respective RecGrant modes at the same time. The last two properties are used to assure that the subsystems are able to return a grant, i.e., whenever the SubSys1 is in mode S1Done, the next transition of the system establishes a state in which Manager is in mode Stable.

The SCR specification for the systems is translated into the list of properties using techniques presented in Chapter 2.

6.2 Extensions to Design Language

Our original PDL deals only with state changes. We need to extend the PDL to include synchronization information. We also need to change semantics of some of the constructs to support the interleaving model of concurrency.

6.2.1 Synchronization Constructs

We add two new constructs to the PDL: <u>Read</u> and <u>Write</u>. These identify points at which mode classes exchange information. Our choice of synchronization semantics was guided by the following considerations:

- Mode classes can react to each other's state changes.
- Synchronization should preserve "freshness" of events, i.e. events should reach other processes before any other local events are allowed to occur.
- Only communication variables need to be passed between mode classes.

We perform variable exchange and synchronization via shared buffers (queues of length one). Using our set-based approximation technique, we give a buffer the following definition:

Definition 6.2 *A buffer is a tuple* $B = \langle L, \texttt{info}, \mathrm{Read}(), \mathrm{Write}() \rangle$*, where*

> *L (a string) is a unique identifier for the buffer;*
>
> `info` *is a set of variable-value pairs* $\{(r_j, v_{r_j})\}$*,* $\forall r_j \in COM$*,*
>
> *indicating information passed through B;*
>
> Read*() is a routine performing reads from B; and*
>
> Write*() is a routine performing writes to B.*

We say that a buffer b is EMPTY when $\forall r_j \in COM, v(B, r_j) = \{\}$.

Read() and Write() routines have the following semantics:

```
Read:
    1.  Wait until buffer is not EMPTY
    2.  Read information in buffer
    2.  Set buffer information to EMPTY
    3.  Return variable-value pairs to caller

Write:
    1.  Wait until buffer is EMPTY
    2.  Store variable-value pairs in buffer
    3.  Return
```

- An annotation "L: Read" indicates that a process reads the values of its inputs from a buffer named L, calling L's Read routine and waiting until it returns values of all communication variables for current process.

- An annotation "L: Write" indicates that a process writes values of its outputs to a buffer named L, calling L's Write routine with values of communication variables and waiting until it returns.

Several processes can write to the same buffer or read from it. However, a single process cannot read from and write to the same buffer.

To guarantee that our model of synchronization is interleaving, we define the semantics of an Update annotation so that no two Updates can occur in the entire system at the same time.

6.2.2 Example: Designing MES

We use a detailed design of our specifications of the mutual exclusion protocol from Section 6.1.1 to illustrate the language and the design technique. Figure 6.2 shows the design of Manager (a)) and SubSys1 (b)). This system has three buffers: A, B and C. The Manager waits at a point "A: Read" until some information (the value

```
    /* Manager */
{ @@ INITIAL Manager=Stable;
  while(1) {
    if (manager is free) {
      @@ Assert Manager=Stable;
      @@ A: READ;
      /* someone requests grant */
      if (this is system two) {
        @@ ASSERT In(SubSys2=S2Wait)=true;
        @@ UPDATE Manager=Grant2;
      }
      else {
        @@ ASSERT In(SubSys1=S1Wait)=true;
        @@ UPDATE Manager=Grant1;
      }
      /* tell processes */
      @@ B: WRITE;
    }
    else{
      /* one subsystem is done */
      @@ Assert ~Manager=Stable;
      @@ C: Read;
      @@ UPDATE Manager=Stable;
    }
  }
}
```

a)

```
     /* subsystem 1 */

{ @@ INITIAL SubSys1=S1Done &
     Time=false & Done=false;
  while (1) {
    /* wait for some time */
    @@ UPDATE Time=true;
    @@ UPDATE SubSys1=S1Wait
       & Time=false;
    @@ A: WRITE;
    @@ B: READ;
    /* assume grant is given */
    @@ ASSERT In(Manager=Grant1)=true;
    @@ UPDATE SubSys1=S1RecGrant
       & Done=false;
    /* work for a while */
    @@ UPDATE Done=true;
    @@ UPDATE SubSys1=S1Done;
    @@ C: WRITE;
  }
}
```

b)

Figure 6.2: Detailed design of MES. Manager and SubSys1 processes.

of In(SubSys1=S1Wait) or In(SubSys2=S2Wait)) is written into buffer A at a point "A: Write" by either of the subsystems. Values of variables In(Manager=Grant1) and In(Manager=Grant2) are written into buffers B and C by Manager, and read by the subsystems. Finally, the values of In(SubSys1=S1Done) and In(SubSys2=S2Done) are written into the buffer C and read by Manager at a point "C: Read".

6.2.3 Discussion

The synchronization mechanism described above is similar to a hand-shaking protocol used in communication software: a process expresses the intention to write, waits for the reader's acknowledgment, sends the data, and waits for an indication that it has been received. Although it allows for correct interpretation of events, this semantics might not be the most appropriate for all applications. In some cases we might want to prioritize messages, while in others we might not want the write to be blocking. Possible semantics include (priority) queues of messages, asynchronous writes into one buffer, rendezvous, and many others. Most of these alternative semantics result in weaker analysis, i.e., they fail to guarantee that events cannot occur in processes until the synchronization completes. However, if we do not allow events to come from different processes, then no such restrictions are necessary.

Enforcing the interleaving model by restricting semantics of Update statements is meant to ensure that no race conditions can occur in the system. We might not need to do this if we assure that mode classes cannot depend on each other for triggering events. Evaluation of design and analysis tradeoffs among different types of buffers and various semantics for process composition needs further investigation.

6.3 Creating an Abstraction

The detailed design is used to create a Concurrent Design Flow Graph ($CDFG$) which is abstracted into a Finite-State Machine (FSM). Afterwards, the FSM is used to check system properties.

6.3.1 Constructing a Flow Graph

Definition 6.3 *A Concurrent Design Flow Graph with Buffers(CDFG) of a detailed design D is a directed graph $\langle V, E, B, V_0 \rangle$, where*

> V *is the set of nodes corresponding to decisions, joins and annotations of* D,
>
> E *is the set of edges representing sequential or synchronization control flow,*
>
> B *is the set of shared buffers used in communication between processes, and*
>
> V_0 *is a set of entry points, one for each process.*

A sequential control flow edge represents sequential flow of control within the design of each process. A synchronization edge is an edge from a node containing "L: Write" to buffer L, and from buffer L to a node containing "L: Read". Since a node can contain either a Read or a Write annotation, a node can have at most one synchronization edge. A buffer, however, can have several incoming and outgoing edges. Let $pred_{seq}(n)$ and $pred_{sync}(n)$ be sets of all predecessors of n (n can be either a node or a buffer) via sequential and synchronization edges, respectively.

Figure 6.3 contains a portion of CDFG corresponding to the code fragment in Figure 6.2. (SubSys2 is omitted for clarity.) Sequential and synchronization edges are identified in the figure following the convention of [GS93]. Computation of system states at each node of the CDFG is performed using dataflow techniques[ASU88, GS93, LC91] similar to those described in Section 3.2. We extend these techniques to include synchronization information.

First, we compute information generated and known at every node. Read and Write nodes do not generate new information, and thus their `gen` and `known` sets are EMPTY. An annotation @@Update Manager=Grant1 indicates that Manager enters a mode Grant1, and sets an output variable In(Manager=Grant1) to true (and, via environmental conditions, In(Manager=Grant2) to false). This annotation produces the following `gen` set:

$$\begin{aligned}\texttt{gen} \quad &= \quad \{(\text{Manager}, \{\text{Grant1}\}), (\text{In(Manager=Grant1)}, \{\text{true}\}),\\ &\qquad (\text{In(Manager=Grant2)}, \{\text{false}\})\}\end{aligned}$$

since In(Manager=Grant1) and In(Manager=Grant2) are the only communication variables set by Manager.

We propagate `gen` and `known`[1] information throughout the graph, computing `in` and `out` sets. Each variable for each buffer node is initialized to its maximum ($\top$) value, i.e., a set containing all possible values for all variables, and the rest of the sets are initialized as discussed in Section 3.2. Our SCR model assumes that each process is data-independent, and explicit synchronization is used for data exchange. Thus, our treatment of synchronization edges is similar to that of sequential edges. We define $\text{pn}(n)$ to indicate a process number of a node n. For a set s, we define a restriction operator $\lhd$:

$$COM \lhd s \equiv \{(r_j, v(r_j))\}, \text{s.t.}((r_j, v(r_j)) \in s) \ \wedge \ r_j \in COM$$

[1]These sets are computed from Assert annotations.

`in` and `out` sets are initialized to {}. `gen` and `known` sets are propagated throughout the DFG via a least fixed point algorithm. To compute `in` of a node, we union the `out` information of its sequential predecessors. For a Read node, we also union `info` of the corresponding buffer node:

$$\texttt{in}(n) = \bigcup_{p \in pred_{seq}(n)} \texttt{out}(p) \bigcup_{b = pred_{sync}(n)} (COM_{in,pn(n)} \lhd \texttt{info}(b))$$

A set F of transfer functions is similar to the sequential case:

Annotation at node n	Transfer function
Initial	$\texttt{out}(\mathrm{n}) = \texttt{gen}(\mathrm{n})$
Update	$\texttt{out}(\mathrm{n}) = \mathrm{repl}(\texttt{in}(\mathrm{n}), \texttt{gen}(\mathrm{n}))$
Assert (single disjunct)	$\texttt{out}(\mathrm{n}) = \texttt{in}(\mathrm{n}) \sqcap \texttt{known}(\mathrm{n})$
buf: Write	$\texttt{out}(\mathrm{n}) = \texttt{in}(\mathrm{n})$
	$\texttt{info}(\mathrm{buf}) = \bigcup_{p \in pred_{sync}(buf)} (COM_{out,pn(p)} \lhd \texttt{out}(p))$
Read or none	$\texttt{out}(\mathrm{n}) = \texttt{in}(\mathrm{n})$

Consider the initial calculation of `in`(6) for SubSys1. The sequential predecessor of this node is 5 (same process), and synchronization is via buffer B's Read operation. The only writer to this buffer is node 12 of Manager. Its predecessor is node 11, and the predecessors of node 11 are 8 and 10. Node 8 contains a transition to Manager(Grant1) and thus sets In(Manager=Grant1) to true and In(Manager=Grant2) to false. Node 10 contains a transition to Manager=Grant2, setting In(Manager=Grant2) to true and In(Manager=Grant1) to false. Information from these two nodes is combined in node 11, propagated to node 12 and then written to the buffer. So,

$$\begin{aligned}\texttt{info}(\mathrm{B}) = \quad & \{(\mathrm{In}(\mathrm{Manager{=}Grant1}), \{\mathrm{true,false}\}),\\ & (\mathrm{In}(\mathrm{Manager{=}Grant2}), \{\mathrm{true,false}\})\}\end{aligned}$$

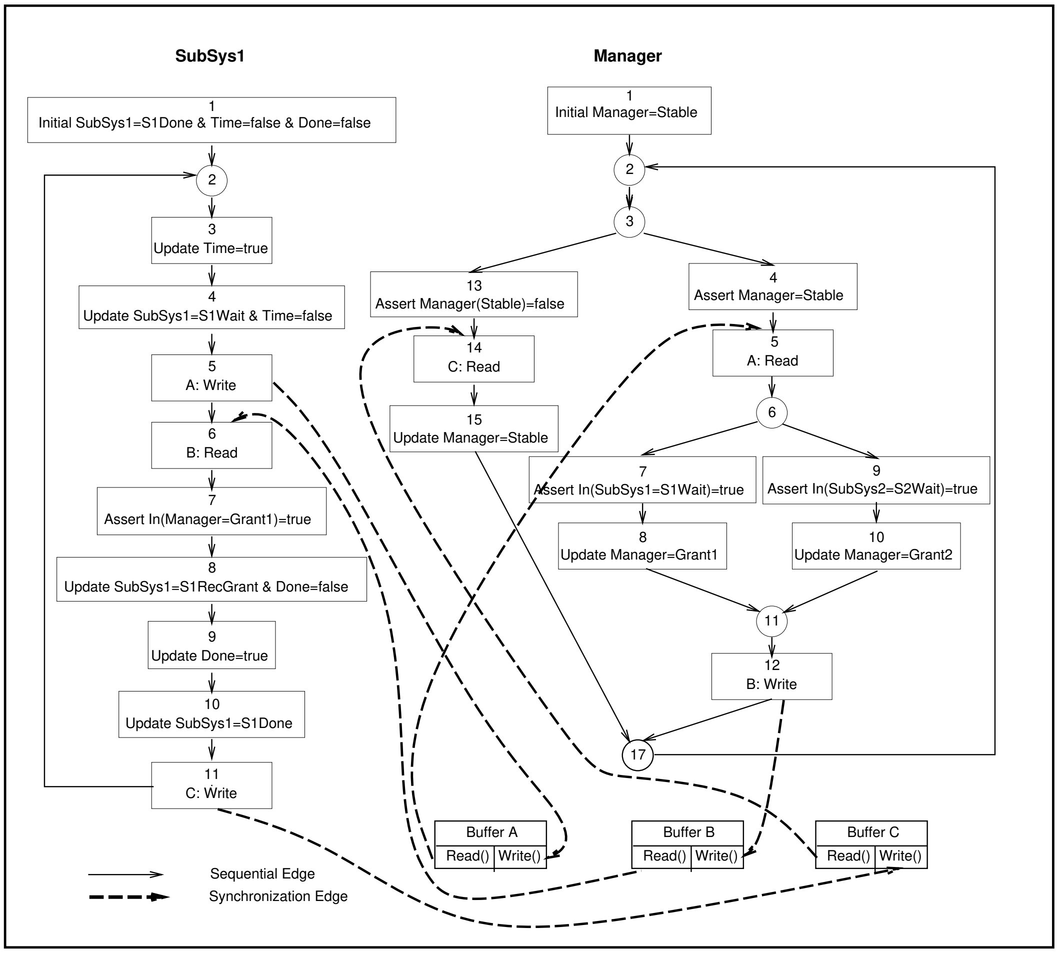


Figure 6.3: Fragment of CDFG of MES.

SubSys1 executes "B: Read" and receives

$$COM \lhd \text{info(B)} = \{(\text{In(Manager} = \text{Grant1)}, \{\text{true}, \text{false}\})\}.$$

This information is unioned with `out`(5) (SubSys1) to obtain

$$\begin{aligned} \texttt{in}(6) \quad = \quad & \{(\text{SubSys1, \{S1Wait\}), (Time, \{false\}), (Done, \{true\}),} \\ & \text{(In(Manager=Grant1), \{true,false\})\}.} \end{aligned}$$

We show `info` sets computed for each buffer below.

Name	Writers	Readers	`info`
A	node 5, SubSys1 & SubSys2	node 5, Manager	{(In(SubSys1=S1Wait), {true,false}), (In(SubSys1=S1Done), {false}), (In(SubSys2=S2Wait), {true,false}), (In(SubSys2=S2Done), {false})}
B	node 12, Manager	node 6, SubSys1 & SubSys2	{(In(Manager=Grant1), {true,false}), (In(Manager=Grant2), {true,false})}
C	node 11, SubSys1 & SubSys2	node 14, Manager	{(In(SubSys1=S1Wait), {false}), (In(SubSys1=S1Done), {true,false}), (In(SubSys2=S2Wait), {false}), (In(SubSys2=S2Done), {true, false})}

After a fixed point is computed, Analyzer removes unreachable nodes, i.e., those nodes for which some variables have {} as their values. However, finding an unreachable node has more serious implications than in the sequential case since unreachable nodes might lead to fewer possible values of communications variables written to buffers, which might affect other processes. For example, assume that the designer made a mistake on line 9 of Manager process, changing Manager state to Grant1. Then information written to buffer B is {(In(Manager=Grant1), {true}), (In(Manager= Grant2), {false})}, and this is what the subsystems read in node 6. SubSys2 assumes that it has been given a grant in node 7 (Assert In(Manager=Grant2)=true) which contradicts this node's `in` information, making this node, as well as nodes 8, 9, 10, and 11 unreachable. This means that SubSys2 never writes to buffer C, which might have an effect on Manager process, etc.

Figure 6.4 summarizes the algorithm used by Analyzer to compute the system information and to reduce the impact of unreachable states.

Algorithm 6

```
initialize in and out sets of every node to EMPTY
initialize each buffer's info to ⊤ for each variable
mark all states as reachable
for each process p
  change[p] = 1
repeat
  while (there is process p s.t. change[p] = 1) {
    for each p {
      if change[p] {
        change[p] = 0
        compute fixed point state information for this process, considering
        only reachable states
        identify all unreachable states in p
        if found new unreachable states {
          notify user
          change[p] = 1
        }
      }
    }
  }
  for each buffer b {
    compute info(b) -union of out sets of all reachable writers to b
    if info(b) has changed
      for all processes p reading from b
        change[p] = 1
  }
until there are no changes in buffers
```

Figure 6.4: Algorithm to compute system information.

6.3.2 Constructing a Finite-State Machine

Since the resulting CDFG contains nodes which do not reflect state changes, we remove all nodes which do not correspond to Initial, Update and Read annotations to construct the Finite-State Machine. Nodes containing Read statements are just place holders indicating that Analyzer should search for Triggering conditions in other processes.

This process of creating a FSM from CDFG is the same as described in Section 3.5. FSM corresponding to the CDFG on Figure 6.3 is shown in Figure 6.5, with Read statements shaded. Nodes 2, 5, 7 and 11 of SubSys1 and SubSys2, and nodes 2, 3, 4, 6, 7, 9, 11, 12, 13, 16, and 17 of Manager are removed. Node 11 was a successor of node 8; but since 11, 12 and 17 were removed, nodes 5 and 14 became the successors of 8. The removed node 12 was connected via a synchronization edge to buffer B and from there to nodes 6 of the subsystems. Thus, node 8 is now connected to the subsystems via synchronization edges.

6.4 Computing Transitions

Computation of transitions and verification of automatically-generated properties for concurrent systems is done similarly to the way described in Chapter 4. In the following two sections, we just point out the differences.

To compute events triggering transitions into node s, we traverse the FSM edges backwards until an Update node n is found on every branch, skipping nodes that do not correspond to Update annotations. To determine if an event occurred, we use information generated at each n and check it against `in`(s) and `out` sets of the predecessors of n. The algorithm to compute transitions leading to a node s in the presence of concurrency, is shown in Figure 6.6. Output of this algorithm, as in the sequential case, is a set of transitions (n, `to`, `from`, `trigger`, `when`).

For example, consider calculating an event which causes the transition to Manager(Stable) at node 15 of Manager process (Figure 6.5). The predecessor of this node is a specially marked Read node (node 14) indicating that the events should come from

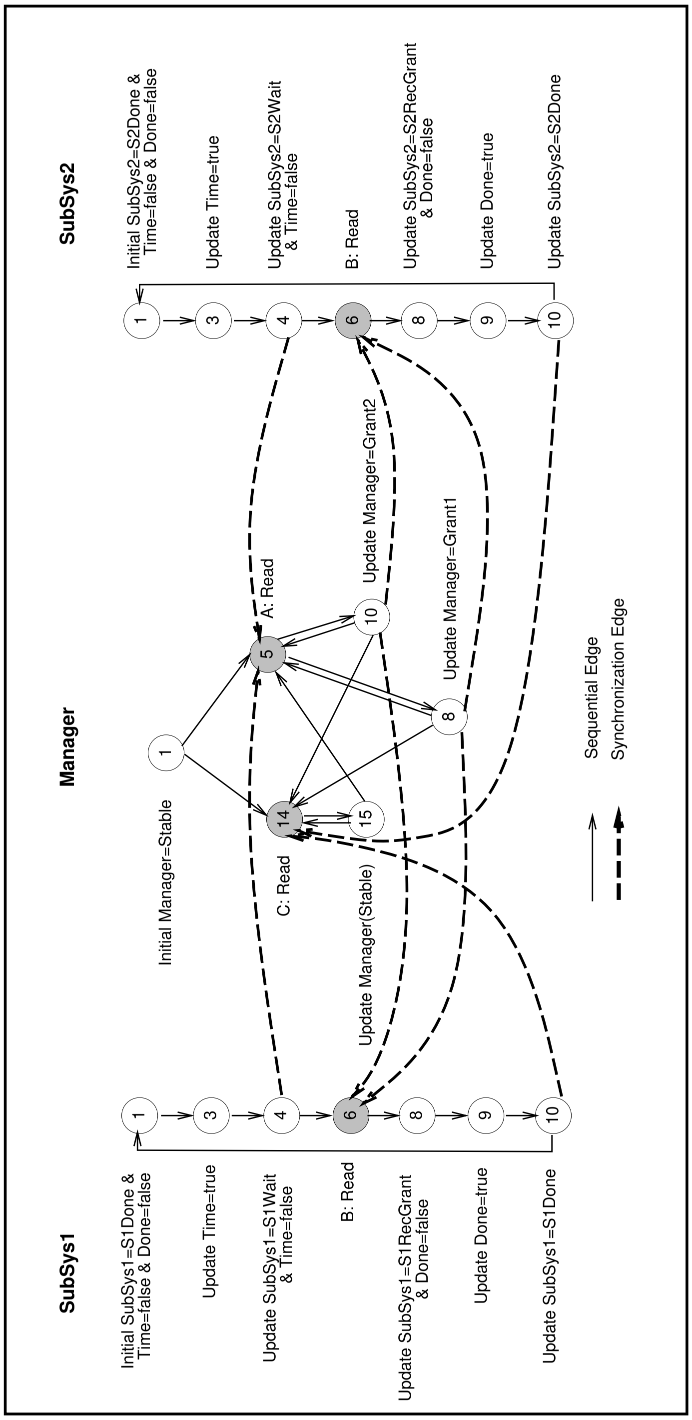


Figure 6.5: FSM of MES.

another process. Thus, Analyzer looks for predecessors of 14, finding nodes 10 in SubSys1 and SubSys2. These nodes contain Update annotations, changing the modes of

Algorithm 7

Inputs: Finite state machine (FSM) and
node s of process i for which to compute the transitions.
Outputs: A set of transitions to node s.
Algorithm:

```
all_transitions = {}
For every n ∈ pred_seq(s) {
    If n contains an Update or an Initial annotation
        compute transitions between n and s as shown in Figure 4.2,
        Chapter 4 and add them to all_transitions.
    If n contains a Read annotation {
        Follow synchronization edges to other processes to build
        a set L of nodes containing Update or Initial annotations.
        For each node l ∈ L {
            triggs = true
            when = out(n) ⊓ in(s)
            For each r_j ∈ (C_pn(l) ∩ M_pn(s)) { /* communication variable */
                triggs = triggs ∧ events(r_j, ⋃_{∀p∈pred(l)} out(p), gen(l) ⊓ when)
                /* values of communication variables should be obtained */
                /* from out(l) */
                when = repl(r_j, when, v(r_j, out(l)) ∩ v(r_j, in(s)))
            }
            /* Rewrite triggs to be in form ⋁_k ⋀_{r_j∈R} exp_{k,j} */
            For each of the k disjuncts d_k of triggs {
                Evaluate d_k interpreting none(r_j) as true.
                If d_k = true, then replace d_k with special value NONE
                trigger[k] = d_k
                when[k] = when
```

Figure 6.6: Algorithm to compute transitions for concurrent processing.

SubSys1 and SubSys2, respectively. For process SubSys1,

$$\begin{aligned}\texttt{gen}(10,\text{SubSys1}) \quad = \quad & \{(\text{SubSys1}, \{\text{S1Done}\}), (\text{In(SubSys1=S1Done)}, \{\text{true}\}),\\ & (\text{In(SubSys1=S1Wait)}, \{\text{false}\})\}\end{aligned}$$

The initial When condition is constructed from $\texttt{out}(14,\text{Manager}) \sqcap \texttt{in}(15,\text{Manager})$ by replacing values of communication variables with their values in $\texttt{out}(10,\text{SubSys1}) \sqcap$

```
/* remove trigger values from WHEN condition */
For each r_j ∈ (C_pn(l) ∩ M_pn(s)) /* communication variable */
    If partof(r_j, d_k) {
        when[k] = repl(r_j, when[k], {})
/* compute to and from values for modeclass and controlled variables */
For each r_j ∈ R_pn(s) − M_pn(s) s.t. v(r_j, gen(s)) ≠ {} {
    /* remove values changed between source and destination nodes */
    when[k] = repl(r_j, when[k], {})
    For each value_m ∈ v(r_j, gen(s)) {
        to = (r_j = value_m) /* starting value of r_j */
        For each value_t ∈ v(r_j, out(n) ⊓ in(s)) {
            from = (r_j = value_t) /* ending value of r_j */
            If value_m ≠ value_t
                /* otherwise there is no transition */
                all_transitions = all_transitions ∪
                        (l, to, from, trigger[k], when[k])
        }
    }
}
```

Figure 6.6: Algorithm for computing events for concurrent processing (cont'd).

---

in(15,Manager):

$$\texttt{when} = \{(\text{In(SubSys1=S1Done)}, \{\text{true}\}), (\text{In(SubSys1=S1Wait)}, \{\text{false}\}), (\text{In(SubSys2=S2Done)}, \{\text{true,false}\}), (\text{In(SubSys2=S2Wait)}, \{\text{false}\}) (\text{Manager}, \{\text{Grant1,Grant2}\}), (\text{In(Manager=Grant1)}, \{\text{true,false}\}), (\text{In(Manager=Grant2)}, \{\text{true,false}\})\}$$

Triggering conditions are constructed from values of communication variables. The only variables used in communication between SubSys1 and Manager are In(SubSys1=S1Done) and In(SubSys1=S1Wait). Node 9 is the only predecessor of node 10.

$$
\begin{aligned}
\text{triggs} &= \text{events(In(SubSys1=S1Done), } \texttt{out}\text{(9,SubSys1),} \\
&\quad \texttt{gen}\text{(10,SubSys1)} \sqcap \texttt{when}) \\[1em]
\texttt{out}\text{(9,SubSys1)} &= \{\text{(SubSys1, \{S1RecGrant\}), (In(SubSys1=S1Done),\{false\}),} \\
&\quad \text{(In(SubSys1=S1Wait), \{false\}), (Done, \{true\}), (Time,} \\
&\quad \text{\{false\}), (In(Manager=Grant1), \{true\})}\}
\end{aligned}
$$

For communication variables, `gen`(10, SubSys1) ⊓ `when` = {(In(SubSys1=S1Done), {true}), (In(SubSys1=S1Wait), {false})}. Functions events(In(SubSys1=S1Done, ...) and events(In(SubSys1=S1Wait), ...) yield @T(In(SubSys1=S1Done)) and none(In(SubSys1=S1Wait)), respectively. So,

$$\text{triggs} = \text{@T(In(SubSys1=S1Done))}$$

The Triggering condition has just one disjunct. On the first iteration of the loop, `to` is (Manager=Stable) and `from` is (Manager=Grant1); on the second iteration, `from` becomes (Manager=Grant2). We replace In(SubSys1=S1Done)'s value with {} in `when`. We also remove values corresponding to Manager from `when` since Manager appears in `gen`(15,Manager). The result is two transitions, I_1 and I_2, for each starting value of Manager.

Processing of transitions triggered by changes in node 10 of SubSys2 is done similarly, resulting in transitions I_3 and I_4. All transitions for node 15 of Manager are shown in Figure 6.7.

6.5 Verification of Automatically-Generated Properties

Verification of automatically-generated properties is done as described in Chapter 4. For example, one of the OLT properties asserts that if Manager is in mode Stable in its next state, then it either must be in that mode already, or one of the requirements'

I_1: Transition from Manager=Grant1 to Manager=Stable:
@T(In(SubSys1=S1Done)) WHEN [{(In(SubSys1=S1Wait), {false}), (In(SubSys2=S2Done), {true,false}), (In(SubSys2=S2Wait), {false}), (In(Manager=Grant1), {true,false}), (In(Manager=Grant2), {true,false})}]

I_2: Transition from Manager=Grant2 to Manager=Stable:
@T(In(SubSys1=S1Done)) WHEN [{(In(SubSys1=S1Wait), {false}), (In(SubSys2=S2Done), {true,false}), (In(SubSys2=S2Wait), {false}), (In(Manager=Grant1), {true,false}), (In(Manager=Grant2), {true,false})}]

I_3: Transition from Manager=Grant1 to Manager=Stable:
@T(In(SubSys2=S2Done)) WHEN [{(In(SubSys2=S2Wait), {false}), (In(SubSys1=S1Done), {true,false}), (In(SubSys1=S1Wait), {false}), (In(Manager=Grant1), {true,false}), (In(Manager=Grant2), {true,false})}]

I_4: Transition from Manager=Grant2 to Manager=Stable:
@T(In(SubSys2=S2Done)) WHEN [{(In(SubSys2=S2Wait), {false}), (In(SubSys1=S1Done), {true,false}), (In(SubSys1=S1Wait), {false}), (In(Manager=Grant1), {true,false}), (In(Manager=Grant2), {true,false})}]

Figure 6.7: Events and transitions discovered for node 15(Manager).

---

transitions is occurring: from Grant1 on an event @T(In(SubSys1=S1Done)) or from Grant2 on @T(In(SubSys2=S2Done)):

$$P_{olt} = \forall n,\ (n \models (\text{Manager=Stable}) \rightarrow (\forall p \in \text{pred}(n),\ (p,n) \models (\text{Manager=Stable}) \ \vee (p,n) \models (\text{@T(In(SubSys1=S1Done))} \wedge (\text{Manager=Grant1})) \vee (p,n) \models (\text{@T(In(SubSys2=S2Done))} \wedge (\text{Manager=Grant2}))))$$

In this property,

$\texttt{dest}_{olt}$	is	Manager=Stable
$\texttt{source}_{olt,1}$	is	Manager=Grant1
$\texttt{trcond}_{olt,1}$	is	@T(In(SubSys1=S1Done))
$\texttt{whcond}_{olt,1}$	is	{(In(SubSys1=S1Wait), {true,false}), (In(SubSys2=S2Wait), {true, false}), (In(SubSys2=S2Done), {true,false}), (In(Manager=Grant1), {true,false}), (In(Manager=Grant2), {true,false})}
$\texttt{source}_{olt,2}$	is	Manager=Grant2
$\texttt{trcond}_{olt,2}$	is	@T(In(SubSys2=S2Done))
$\texttt{whcond}_{olt,2}$	is	{(In(SubSys1=S1Wait), {true,false}), (In(SubSys2=S2Wait), {true, false}), (In(SubSys1=S1Done), {true,false}), (In(Manager=Grant1), {true,false}), (In(Manager=Grant2), {true,false})}

For all nodes other than 1 and 15 of Manager, the `to` part of transitions is not equal to Manager=Stable, so P_{olt} holds vacuously. Node 1 is the starting node for Manager process, so it has no predecessors and thus P_{olt} holds. Transitions generated for node 15 of Manager are I_1-I_4, as shown in Figure 6.7. The `from` part of I_1 = $\texttt{source}_{olt,1}$, $\texttt{trigger} \rightarrow \texttt{trcond}_{olt,1}$ and $\texttt{when} \rightarrow \texttt{whcond}_{olt,1}$. So, no errors are reported for I_1. Similarly, no errors are reported for I_4. However, Analyzer reports errors for I_2 and I_3, as shown in Figure 6.8.

One of the ALT properties automatically generated from the requirements indicates that there exists a state in which Manager is in Grant1 and an event @T(In(SubSys1 =S1Done)) occurs, and Manager is Stable in one of the next states:

$$\begin{aligned} P_{alt,1} \quad = \quad & \exists n,\ \exists p \in \text{pred}(n),\ n \models (\text{Manager=Stable}) \ \wedge \\ & (n,p) \models ((\text{Manager=Grant1}) \wedge \ @\text{T}(\text{In}(\text{SubSys1=S1Done}))) \end{aligned}$$

Another ALT property is that there exists a state in which Manager is in Grant2 and an event @T(In(SubSys2=S2Done)) occurs, and Manager is Stable in one of the next states:

$$\begin{aligned} P_{alt,2} \quad = \quad & \exists n,\ \exists p \in \text{pred}(n),\ n \models (\text{Manager=Stable}) \wedge \\ & (n,p) \models ((\text{Manager=Grant2}) \ \wedge \ @\text{T}(\text{In}(\text{SubSys2=S2Done}))) \end{aligned}$$

```
                    ANALYZER (Version 3.01)

Found an event on line 13, process SubSys1.
Error on line 22, process Manager, in mode class MANAGER:
   illegal transition from mode(s) {Grant2} to mode(s) {STABLE}.

Event is @T(In(SubSys1=S1Done)) WHEN [In(SubSys1=S1Wait)={false} &
   In(SubSys2=S2Done)={true,false} & In(SubSys2=S2Wait)={false} &
   In(Manager=Grant1)={true,false} & In(Manager=Grant2)={true,false}]


Found an event on line 13, process SubSys2.
Error on line 22, process Manager, in mode class MANAGER:
   illegal transition from mode(s) {Grant1} to mode(s) {STABLE}.

Event is @T(In(SubSys2=S2Done)) WHEN [In(SubSys2=S2Wait)={false} &
   In(SubSys1=S1Done)={true,false} & In(SubSys1=S1Wait)={false} &
   In(Manager=Grant1)={true,false} & In(Manager=Grant2)={true,false}]
```

Figure 6.8: Output of analysis of MES.

Transitions generated for node 15 of Manager, which is reachable from the start state, enable the algorithm to mark $P_{alt,1}$ and $P_{alt,2}$ as satisfied. $P_{alt,1}$ is satisfied by I_1 and $P_{alt,2}$ is satisfied by I_4.

6.6 Verification of System Goals

To verify system goals, otherwise called user-defined properties, we remove nodes corresponding to Read annotations and concurrent edges from FSM, breaking the global FSM into a set of FSMs, one for each process. Each system goal is verified in a separate traversal of one or more FSMs. First, Analyzer determines processes on which it is sufficient to verify a particular property: A process i needs to be considered during verification of a property U_j if some variable of i (monitored, controlled, communication or mode class) is part of U_j. The number of states to be used for the verification is identified conservatively, i.e., no other states need to be considered. If a property can be verified on just one process, it is considered local. Otherwise, it is global.

6.6.1 Verification of Local Goals

Local properties, like

reach (Manger=Grant2)

which involves only Manager process, are verified using techniques described in Section 4.7. To verify this property, Analyzer searches FSM of Manager process for a state where it is in mode Grant2, finds one (say, node 10) and decides that this property holds. As noted earlier, reach properties are verified optimistically, so it is possible that Analyzer reports no violation although the property does not hold in the design. However, when we try to verify the same property on the incorrect design (see Section 6.3.1, where we changed the annotation at node 10 to @@Update Manager=Grant1), Analyzer reports an error. This means that the error is present in the actual design.

6.6.2 Verification of Global Goals

During verification of global properties, we identify participating processes for each property P and compute a composition of reachable states of each participating process. We assume that a reachable state of one process can be simultaneously executable with any other reachable state of another process (see Section 6.6.3 below). These composed states become a model for verification of P. Let R_i be a set of variables (modeclass, monitored, controlled and communication) for process i.

A composition of nodes n_1 and n_2, written as (n_1, n_2) is

$$
\begin{aligned}
(n_1, n_2) \quad = \quad & \{(r_j, \mu_j) \mid (r_j \in R_{\mathrm{pn}(n_1)} \wedge r_j \notin R_{\mathrm{pn}(n_2)} \wedge v(r_j, \mathtt{out}(n_1)) = \mu_j) \vee \\
& (r_j \notin R_{\mathrm{pn}(n_1)} \wedge r_j \in R_{\mathrm{pn}(n_2)} \wedge v(r_j, \mathtt{out}(n_2)) = \mu_j) \vee \\
& \mu_j = v(r_j, \mathtt{out}(n_1)) \sqcup v(r_j, \mathtt{out}(n_2))\}
\end{aligned}
$$

i.e., $(n_1,\ n_2)$ consists of all variable-value pairs of `out` sets of these nodes for variables local for $\mathrm{pn}(n_1)$ and $\mathrm{pn}(n_2)$, and unions of values of variables shared between $\mathrm{pn}(n_1)$ and $\mathrm{pn}(n_2)$.

Consider computing a composition of nodes 9 of SubSys1 and 15 of Manager.

`out`(9, SubSys1)	=	{(SubSys1, {S1RecGrant}), (In(SubSys1=S1Done), {false}),(In(SubSys1=S1Wait), {false}), (Done, {true}), (Time, {false}), (In(Manager=Grant1), {true})}
`out`(15, Manager)	=	{(Manager, {Stable}), (In(Manager=Grant1),{false}), (In(Manager=Grant2), {false}), (In(SubSys1=S1Done), {true,false}), (In(SubSys2=S2Done), {true,false}), (In(SubSys1=S1Wait), {false}), (In(SubSys2=S2Wait), {false})}
(9, 15)	=	{(SubSys1, {S1RecGrant}), (Done, {true}), (Time, {false}), (Manager, {Stable}), (In(Manager=Grant1), {false}), (In(Manager=Grant2), {false}), (In(SubSys1= S1Done), {true,false}), (In(SubSys2=S2Done), {true, false}), (In(SubSys1=S1Wait), {false}), (In(SubSys2= S2Wait), {false})}

During verification of the mutual-exclusion property

$$\neg(\text{SubSys1=S1RecGrant} \wedge \text{SubSys2=S2RecGrant}),$$

we determine that it needs to be verified on FSMs for SubSys1 and SubSys2. There are no variables shared between these processes. The composition includes just the local variables for each process. Analyzer goes through composed states and looks for those satisfying SubSys1=S1RecGrant $\wedge$ SubSys2=S2RecGrant namely, (8, 8), (8, 9), (9, 8) and (9, 9), and reports the violations.

6.6.3 Discussion

Processes share communication variables, and it would be appealing to define states to be simultaneously executable if their communication variables have common values in these states. However, processes update values of their communication variables only at points of communication, and for the remainder of processing, these values are maintained in a "cache". In our model, no provisions are made to maintain cache consistency, i.e., when one process changes its mode, it does not send updated versions of communication variables to other processes which use this mode information. Our model was created to facilitate efficient exchange of events between processes, and it does not allow for effective elimination of composed states which cannot exist in the actual design. So, our analysis is overly pessimistic (or, for reach properties, overly optimistic).

6.7 Summary, Limitations and Analysis

In this chapter, we described extensions to Analyzer which enable verification of consistency between concurrent program designs and their requirements. We extended SCR requirements to be able to reason about concurrency. As in the sequential case, we use SCR requirements to generate CTL$\prime$ formulas corresponding to our notion of consistency. We also extended our PDL to include synchronization constructs. Our tool starts with control-flow graphs of each process and then propagates information about each variable throughout the processes without building the global reachability graph. Model-checking techniques are then used to verify that the resulting finite-state machine is a model of CTL′ formulas. Our major contributions to analysis of concurrency are summarized below:

- Our approach keeps track of values of variables as opposed to just communication information, which allows us to determine unreachable states and see if these states affect the dataflow between processes.
- We build a finite-state machine which consists of at most

  $$\Sigma n_i, 1 \leq i \leq |MD|$$

  nodes, where n_i is a number of Update and Read annotations in a detailed design of process i. Verification of all automatically-generated properties is done on this machine.
- During verification of user-defined properties, it is sufficient to look at the composition of only the processes whose variables are used in these properties.

However, this approach works well only with systems with complicated control-flow and infrequent communications. Our modeling of synchronization is rather awkward, and attempts to look at typical benchmark problems, e.g., the gas station example [HL85], did not result in interesting analyses. These problems are stripped of all but the communications information (rendezvous between Ada processes in [HL85]), and thus Analyzer does not perform very well.

To verify a wider range of systems, we would need to make several modifications to our approach:

- Many concurrent applications consist of several identical processes. We need to be able to specify requirements for these systems using SCR without enumerating all of them. It would be more convenient to use arrays of modes and mode classes, as in [JLHM91, Har87].

- A useful and general semantics of communication between processes needs further investigation. We would not have to forbid simultaneous Updates to avoid race conditions if we required that SCR variables have a partial order of dependencies i.e., two mode classes cannot receive events from each other. [Hei96] suggested that processes should not react to events in other processes at all, but only treat them as WHEN conditions. This restriction would enable us to relax our requirement of "freshness" of events, implemented via blocking reads and blocking writes, and permit us to include a wider variety of synchronization primitives.

Trying to extend SCR to look at concurrency violates the black-box nature of the SCR method. SCR has been designed to specify reactive systems; problems without significant computation may be too far from SCR's original application domain.

Chapter 7

Conclusion

7.1 Summary

This dissertation described a method for checking if detailed designs of programs correspond to their requirements. To verify that a system design is consistent with its requirements, we ensure that the design's state transitions are enabled by the same events as those of the requirements, and the requirement's safety properties also hold in the design. To judge global properties like these, it is necessary to determine the possible system states which exist at different program points. The detailed bookkeeping necessary to do this exceeds the capabilities of human reviewers for all but small implementations. We have developed a language for specifying detailed designs, an analysis technique to create a model of a design through abstract interpretation of the language constructs, and a method to automatically generate and check properties derived from requirements to ensure a design's consistency with them. These ideas are implemented in a tool called Analyzer.

The first part of this dissertation deals with verifying the consistency of designs of sequential programs. A requirements specification is translated into a list of first order logic properties that capture the desired notion of consistency. To prove that a design is consistent with its requirements, we demonstrate that the design is a model of all these properties. To create a design, programmers use a mixture of control-flow constructs and annotations - special statements which describe changes in a system's state. These changes involve local (rather than invariant) information and therefore are relatively easy to specify. We use existing dataflow analysis techniques to create a finite state abstraction of the detailed design. This constitutes the necessary components for model checking - an automated technique for verifying that a set of logic formulas holds in a finite-state machine. However, since the finite-state machine was built using static analysis of the design, we cannot remove all infeasible paths from consideration, and thus cannot determine which of the computed events can exist in the design or verify properties exactly, i.e., claim that a property is violated if and only if the analysis reports a violation. We compute an upper bound of the events present in the design and divide the properties into two categories - properties quantified over all system states and properties quantified over some system states. We verify the former pessimistically, i.e., if analysis reports no violations then we know that these properties hold in the design. We verify the later optimistically, i.e., if analysis reports any violations of such properties, then we know that they do not hold in the design.

In the second part of this dissertation, we extend the analysis technique to verify designs of concurrent systems. To facilitate analysis of concurrent designs, we extend the requirements notation to formalize the semantics of communicating mode classes and change the design notation to include communication information. We also extend dataflow techniques to enable computation of state information (including values of variables) without building a system's reachability graph.

7.2 Methodology

The methodology for verifying that properties specified in the requirements are preserved in detailed designs consists of the following steps:

1. A (human) requirements designer provides an SCR tabular specification of the system's required behavior, one transition table for each modeclass and each controlled variable, using the graphical interface of Analyzer (see Appendix A.1).

2. Analyzer's user interface automatically translates the tabular specifications, the condition relationships and the global properties into an ASCII representation acceptable by the analysis engine (see Appendix A.2).

3. Tools like tcart [Atl92] are used to verify that the global properties hold in the requirements.

4. A (human) designer provides a detailed design of the system using a special program design language (see Appendix B).

5. The analysis engine uses requirements specifications to produce a set of first order logic formulas capturing the notion of consistency, uses the design to produce a finite-state machine, and finally applies model-checking techniques to show that the machine is a model of the first order logic formulas. Outputs of the model-checker are lines of the design where Analyzer found a violation of a particular property.

We are assuming a Waterfall model of the development life-cycle, i.e., we think that once the requirements are proven correct, they will not change significantly, and most of the iteration will occur between the last two steps of the process described above.

7.3 Successes

Major contributions of our work are:

- We defined a notion of consistency between SCR-style requirements and other program artifacts. Precise semantics of SCR requirements allowed translation of this notion into a set of first order logic properties.

- We looked at ways of implementing SCR requirements and developed a simple program-design language allowing us to reason about changes of state in terms of requirements variables.

- We developed a technique for building a finite-state machine from user-written detailed designs. State folding approaches (i.e. set-based approximation) allowed us to construct small machines.

- We gave precise definition of how results of verification of properties should be interpreted, by separating the properties into pessimistic and optimistic categories and tailoring our verification to ensure that no properties are checked inconsistently.

- We performed a case-study in which we reverse-engineered an existing implementation of a Water-Level Monitoring System, and compared the requirements of the system with its detailed design. We found a number of important defects, missed during testing of the original implementation.

- We extended SCR to be able to specify concurrent systems.

- For these systems, we developed techniques for computing finite-state abstractions on which we can verify the majority of properties, without having to build a reachability graph.

Our static analysis technique provides fast but imprecise analysis and tells the user exactly how to interpret results of this analysis. This technique reuses formal requirements in the analysis of designs, providing additional incentive for the creation of such requirements. Our technique proved useful on small but realistic systems, and we believe that it will perform well on larger examples.

Although it uses a number of SCR concepts, we believe that our analysis methodology can be easily extended to deal with artifacts developed from other formal requirements notations.

7.4 Limitations

Our approach has a number of important shortcomings:

- The language for user-specified safety properties is very restrictive.
- Our PDL is primitive and not very natural. Variables can be only boolean or enumerated types, and no data structures are allowed.
- The technique does not support a notion of hierarchical verification, i.e., verifying an abstraction and then refining it.
- Our analysis is inexact, i.e., tool identifies places in the design where an error <u>might</u> occur or a property <u>might</u> be satisfied, but users need to confirm analysis results.

- Our technique does not process timing information, essential for dealing with real-time systems.

7.5 Related Work

Analyzer contains features similar to those in several other static analysis systems. To simplify the verification of properties of programs, these systems restrict the forms of their formal specification notations or create abstract models from programs that could be analyzed with state-exploration rather than theorem-proving techniques.

In Inscape[Per87, Per89a, Per89b], complex logical formulas are abstracted to simple predicates which may be primitive or defined in terms of other predicates (like our relationships). Predicates form pre- and postconditions used to specify implementations. A programmer constructs an implementation with an editor that analyzes the implementation's control flow and operation invocations to calculate its pre- and postconditions. During the calculation, Inscape uses pattern matching and simple deduction to determine if the precondition of an operation has been satisfied before its invocation. If not, unsatisfied predicates are propagated backwards through the control-flow graph until Inscape finds operations satisfying them. The predicates of an operation's postconditions are propagated forward through the graph so that they might satisfy a subsequent operation's precondition. To determine if an implementation is correct, Inscape compares the calculated and the specified conditions.

Quick Defect Analysis (QDA)[How90, HW94] also uses a simplified specification language. Partial specifications called hypotheses are embedded in comments to describe properties that objects should have at particular program points. Other comments contain assertions about properties of objects. An interpreter builds an abstract model of the implementation from the assertions and the implementation's control flow graph. Hypotheses are verified with respect to this model. More recent work[HS95] enriches QDA's specification language so assertions also describe event occurrences and hypotheses assert that the implementation's events occur in certain sequences.

The Cecil specification language permits the description of sequencing constraints on user-definable program events (e.g., definitions or uses of variables, operation invocations, etc.) by anchored, quantified regular expressions (AQREs)[OO89, OO90, OO92]. After a user specifies a mapping from programming language constructs to Cecil events, the Cesar analyzer uses dataflow analysis techniques to determine if the implementation meets Cecil constraints.

Aspect's[Jac92, Jac93a, Jac93b] specification notation permits users to write pre- and postconditions about the data dependencies of an operation. Dataflow analysis is used to compute an upper bound on the data dependencies of the implementation. If an asserted dependency is missing, an error is reported.

Clarke et al. [CGL92] also create abstract, finite state models of programs, and use model checking techniques to verify formulas. Programs written in a special finite-state programming language are translated into relational expressions characterizing the program's initial state and transition relation. To reduce the size of the model, users define mappings of implementation values to abstract values and symbolically execute operations on the values. The model checking approach is pessimistic for formulas expressed in $\forall$CTL*[GL91], a subset of CTL in which only universal path quantification is allowed. The authors also identify a large class of temporal formulas for which the verification results are exact, i.e., formulas hold in the model iff they hold in the original program.

7.6 Future Work

In the future, we are planning to address some of the limitations of our approach as well as to look at useful extensions of our work. We are planning to extend the language of user-specified safety properties to include arbitrary first-order logic statements.

Automating consistency checks between code and design statements is another promising approach. This can be achieved by connecting requirements and program variables. The user might provide declarative information describing the dependencies between implementation and requirements variables. We hope that this approach will enable us to find errors of omission, when the annotations are correct but the code is incorrect or missing.

Our experience writing detailed designs shows that the initial version of the design can be automatically generated from the requirements. However, it is usually necessary to hand-optimize the design to "factor out" common processing. We feel that the appropriate method is to generate the design first, let the user optimize it, and then check the resulting design for consistency with the requirements. The resulting design is a good starting point for generating code. The user can provide mappings between program and requirements variables, enabling us to generate code with a similar layout to the design. The generation step can keep design information as an annotation language, allowing for easy commenting and for automatic checking that user modifications to the code preserve its consistency with its requirements.

Since development of correct designs is an interactive process, at times the user might want to make changes to just one modeclass or one input or output device. Developing techniques which check consistency of a slice of a design and a requirements document (e.g., one table or a set of monitored variables) would enable fast analysis of just properties of interest rather than the whole design.

Another interesting direction is to explore the use of hybrid techniques. Our tool identifies locations where pessimistic properties might fail and where optimistic properties might be satisfied. Testing can then be used to determine if these states are reachable. This technique should even be able to suggest test cases that would test the reachability.

We also need to support a richer type system than just booleans and enumerated types, to make our PDL more expressive and more natural. This would also mean extending our techniques of building a finite-state machine.

Appendix A

Requirements Input Language

The SCR requirements analyzed in this dissertation were presented as tables. This appendix describes the front end of Analyzer which provides a graphical interface for specifying the requirements. The front-end produces an ASCII representation from these requirements. The grammar is LALR(1), which enables the use of Lex and Yacc tools for automatic parsing. The grammar is described using BNF notation. All non-terminals are in italics and all terminals are in plain or boldface type. Syntactic units surrounded by brackets [] are optional. Syntactic units surrounded by starred brackets []$*$ may appear zero or more times, and those surrounded by []$+$ may appear one or more times. The rest of this appendix is organized as follows: Section A.1 describes the front-end to Analyzer and uses SWLMS to show how the interface looks like. Section A.2 gives the BNF grammar of SCR. This grammar was largely borrowed from [Atl92] and modified to process controlled variables. Finally, Section A.3 gives ASCII representations of requirements of systems described in this dissertation.

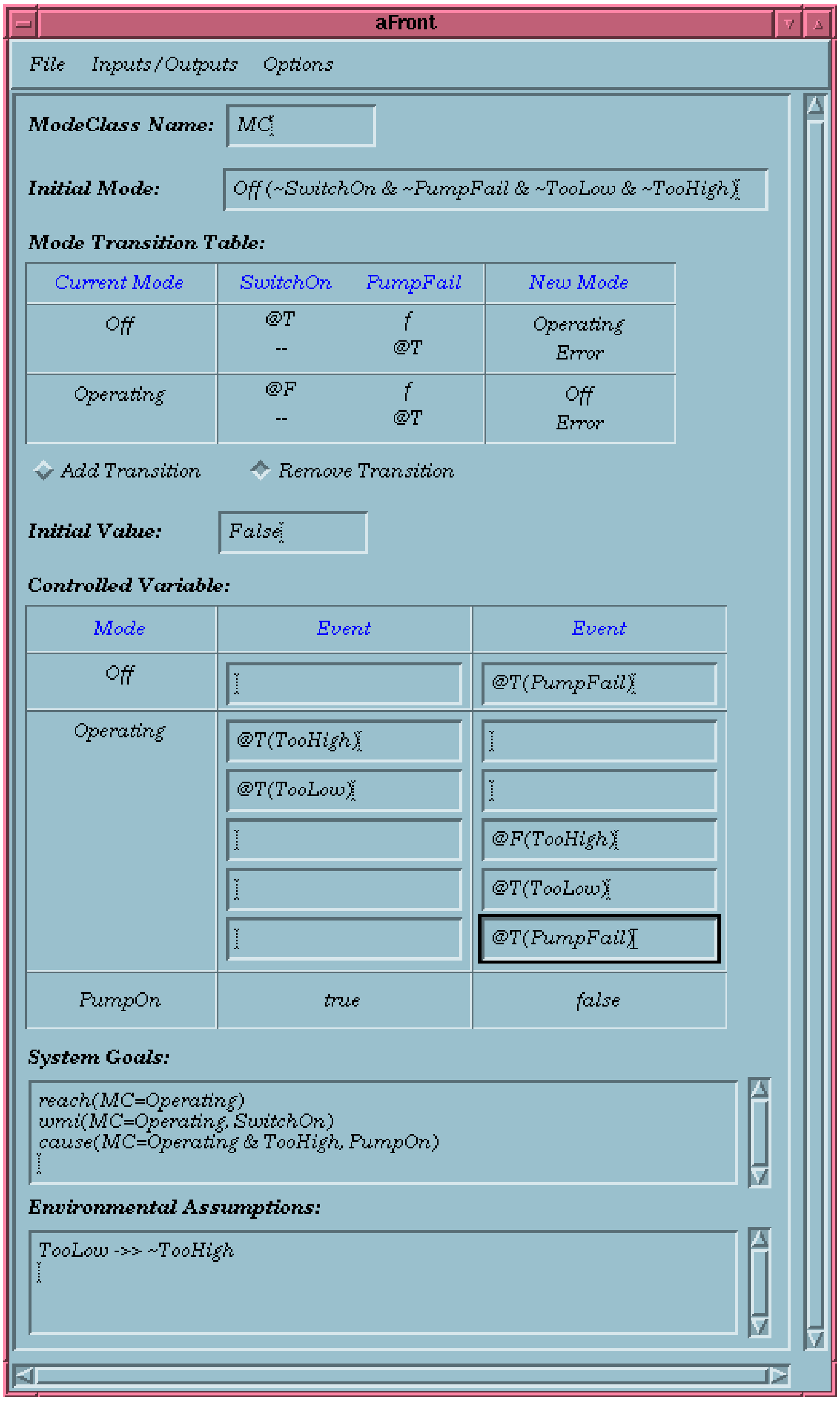
A.1 Tool Interface

Front-end for Analyzer was created to specify requirements for systems to be processed. It brings up a template which the user needs to fill up, allows saving and retrieving the specifications, and generates ASCII representation of the specification, which can then serve as input to Analyzer. The template consists of the following fields:

ModeClass Name	Name of the modeclass to be specified
Initial Mode	Initial condition for the modeclass
Mode Transition Table	For description, see below
Initial Value	Initial value for a controlled variable
Controlled Variables	Changes of values of a controlled variable
System Goals	Described in detail in Section A.2.4
Environmental Assumptions	Described in Section A.2.3.

A menu under Inputs/Outputs allows declaring modes (each declaration automatically adds rows to mode transition and controlled variable tables), monitored variables (each declaration automatically adds columns to the mode transition table) and controlled variables. Selecting Add Transition and Remove Transitions allows adding and deleting rows from both tables. Each entry of mode transition tables toggles between "–", "t", "f", "@T", and "@F", allowing the user to specify Triggering and When conditions for each mode transition. Events triggering changes of values of controlled variables are entered manually.

A snapshot of the front-end of Analyzer depicting a requirements specification of SWLMS is shown below.


aFront
File Inputs/Outputs Options
ModeClass Name: MC
Initial Mode: Off (~SwitchOn & ~PumpFail & ~TooLow & ~TooHigh)
Mode Transition Table:
Current Mode
SwitchOn
PumpFail
New Mode
Off
@T
f
Operating
--
@T
Error
Operating
@F
f
Off
--
@T
Error
Add Transition
Remove Transition
Initial Value: False
Controlled Variable:
Mode
Event
Event
Off
@T(PumpFail)
Operating
@T(TooHigh)
@T(TooLow)
@F(TooHigh)
@T(TooLow)
@T(PumpFail)
PumpOn
true
false
System Goals:
reach(MC=Operating)
wmi(MC=Operating, SwitchOn)
cause(MC=Operating & TooHigh, PumpOn)
Environmental Assumptions:
TooLow ->> ~TooHigh

A.2 ASCII Representation of SCR Requirements

An SCR specification begins with a list of declarations for monitored and controlled variables.

cond-decls	::=	*monitored-decls controlled-decls*
monitored-decls	::=	**MONITORED** *vars*
controlled-decls	::=	**CONTROLLED** *vars*
vars	::=	*cond-name* [, *cond-name*]∗

Conditions are restricted to boolean data types. They can represent environmental conditions or state information. A *state condition* specifies whether or not the system is in a particular mode (see Chapter 2). The format for state conditions is

In(*mode-class*=*mode-name*)

where *mode-class* is the name of a modeclass, and *mode-name* is the name of one of the modes of *mode-class*. So, the format of conditions is

cond-name	::=	*id*
	||	**In**(*mode-class*= *mode-name*)
mode-name	::=	*id*
mode-class	::=	*id*

The list of variable declarations is followed by mode transitions, changes of values of controlled variables, environmental assumptions and system goals. For concurrent requirements, a list of global system goals may be provided in a separate file. The rest of this Appendix describes syntax for these specifications.

A.2.1 Modeclass Declarations

Following the list of controlled variable declarations is a list of one or more modeclass declarations. For concurrent requirements, only one modeclass declaration per specification is allowed. Each modeclass declaration must include an initial state declaration followed by the transition description. A modeclass declaration can have multiple initial modes, but the initial conditions of the initial modes must be disjoint.

class-decls	::=	**MODECLASS** *id initial-mode-decl* [*mode-decl*]+
initial-mode-decl	::=	**INITIAL** *initial-mode-list*
initial-mode-list	::=	*mode-name* (*condition*) [, *mode-name* (*condition*)]*
condition	::=	*condition* & *condition*
	||	*condition* | *condition*
	||	∼ *condition*
	||	(*condition*)
	||	*cond-name*

where *cond-name* refers to the name of a monitored variable. Each mode declaration contains a list of transitions leaving the mode.

mode-decl ::= **MODE** *mode-name* [*trans-decl*]*

Transition declarations specify the destination mode (the source mode is the mode being declared) and the event that triggers the transition.

trans-decl	::=	*mode-name triggered-cond* [*when-cond*]
triggered-cond	::=	**@T** (*cond-name*)
	||	**@F** (*cond-name*)
when-cond	::=	**WHEN** [∼] [*condition*]

A transition event can only have a single triggering condition (we do not allow specification of multiple Triggering conditions like @T(a) & @T(b)[1]), although it can be compound (i.e., @T(a & b) is allowed). An event's When condition, if any, is specified by a special identifier WHEN followed by a formula delimited by brackets []. The formula can be negated.

condition in *triggered-cond* can be an arbitrary expression on monitored and modeclass variables, but not controlled variables. In *when-cond*, though, *condition* can consist of all variables.

A.2.2 Controlled Variables Declarations

Changes of values of controlled variables are done similarly to mode transitions. Each controlled variable declaration must include an initial condition followed by a sequence of one or more value changes. Each variable declaration contains a list of transitions to values true and false.

contr-vars	::=	**CONTROLLED_VARIABLES** [*contr-var*]
	||	
cont-var	::=	**VARIABLE** *id value* [*var-decl*]+
value	::=	**True**
	||	**False**
var-decl	::=	*value triggered-cond* [*when-cond*]

[1]Environmental assumptions allow specifications to have multiple Triggering conditions. We can automatically determine when such a condition is appropriate, but do not allow the user to specify multiple conditions directly.

triggered-cond and *when-cond* have been described in the previous subsection. Since controlled variables can depend on monitored variables, modeclasses and other controlled variables, *conditions* in description of *triggered-cond* and *when-cond* can be any variables in the system.

A.2.3 Environmental Assumptions

Environmental assumptions are specified after all controlled variable tables in the SCR specification. There are five types of assumptions (equality, implication, strict implication, timeline and enumeration), described in detail in Chapter 2. Assumptions can connect only controlled or only monitored variables. All variables have to be declared in this SCR specification. Description of environmental assumptions, or relationships, as we call them, follows below. If there are no condition relationships, then the relationships-decl is empty.

relationships-decl	::=	**RELATIONSHIPS** [*rel-decl*]+
	||	
rel-decl	::=	*cond-name* = *and-list*
	||	*cond-name* -> *and-list*
	||	*cond-name* ->> *and-list*
	||	*cond-name* [< *cond-name*]+
	||	*cond-name* [| *cond-name*]+
and-list	::=	*cond-name* [& *cond-name*]*

If *and-list* specifies a conjunction of conditions, then the first condition in the assumption is related to each of the conditions listed in the *and-list*. Relationship among the conditions listed in the *and-list* is not specified.

A.2.4 System goals

A user can specify a number of system goals to be verified in the design. All of these are safety properties. There are five types of such properties (strong invariant, weak invariant, cause, strict cause and reach), which have been described in detail in Chapter 2. A specification may consist of no user-specified properties, in which case *prop-list* is empty.

prop-list	::=	**PROPERTIES** [*prop-decl*]+
	||	
prop-decl	::=	**smi** (*prop-event* [, *prop-event*])
	||	**wmi** (*prop-event*, *prop-event*)
	||	**reach** (*prop-event*)
	||	**cause** (*prop-event*, *prop-event*)
	||	**strcause** (*prop-event*, *prop-event*)

Strong invariant (smi) has two acceptable formats associated with it. Two arguments mean that in all states where the first *prop-event* is satisfied, the second *prop-event* is also satisfied. Typically, the two-argument format is used to indicate a mode invariant, in which case the first argument is *mode-class=mode-name*. One argument means that *prop-event* should be satisfied in all states.

In requirements for sequential systems, global properties have to be specified in the same file as the rest of the specification. In requirements for concurrent systems, properties can be specified in two locations. A set of "local" properties, i.e., those that are meant to be verified on a single modeclass process, are listed in the same file as the specification for that modeclass. Properties to be verified on the graph of the entire system are specified in a separate file (included in the command line for the tool) and have the following format:

global-prop-list ::= **COMBINED_PROPERTIES** [*prop-decl*]+

||

A.3 Specifications of Systems

This section lists ASCII representations for requirements of systems described in this dissertation. Figure A.1 corresponds to the requirements of the Simple Water-Level Monitoring system, described in Tables 2.3 and 2.4 of Chapter 2. Figure A.2 corresponds to the requirements of the Water-Level Monitoring System described in Chapter 5. Modeclass Normal and controlled variable LowWindowOn were listed in Tables 5.1 and 5.2, respectively. The rest of this section describe requirements of the Mutual Exclusion System (MES) analyzed in Chapter 6. Figure A.3 lists requirements of SubSys1, shown in Table 6.1; Figure A.4 lists requirements of Manager, shown in Table 6.2, and Figure A.5 lists global properties for the MES example.

```
    MONITORED
      SwitchOn, PumpFail, TooHigh, TooLow

    CONTROLLED
      PumpOn

    MODECLASS MC
      INITIAL Off (~SwitchOn & ~PumpFail & ~TooHigh & ~TooLow)
      MODE Off
        Operating @T(SwitchOn) WHEN [~PumpFail]
        Error @T(PumpFail)
      MODE Operating
        Off @F(SwitchOn) WHEN [~PumpFail]
        Error @T(PumpFail)

    CONTROLLED_VARIABLES
VARIABLE PumpOn
        Initial False
        True @T(TooHigh) WHEN [MC=Operating]
        True @T(TooLow) WHEN [MC=Operating]
        False @F(TooHigh) WHEN [MC=Operating]
        False @F(TooLow) WHEN [MC=Operating]
        False @T(PumpFail) WHEN [MC=Operating]
        False @T(PumpFail) WHEN [MC=Off]

    RELATIONSHIPS
      TooHigh ->> ~TooLow

    PROPERTIES
      wmi (MC=Operating, SwitchOn)
      wmi (MC=Off, ~SwitchOn)
      reach (MC=Operating)
      cause (MC=Operating & TooHigh, PumpOn)
```

Figure A.1: Requirements for Simple Water-Level Monitoring System.

```
MONITORED
  InsideHysR, WithinLimits, ShutdownLockTime200, SlfTstPressed,
  SlfTstPressed500, ResetPressed, ResetPressed3000, InTest0,
  InTest2000, InTest4000, InTest14000, FlashOff5, ControlUnitFail,
  TimeDeviceFail, LevelDeviceFail, LevelHigh, LevelLow, PowerFail

CONTROLLED
  AlarmOn, HighWindowOn, LowWindowOn, PumpSwitchOpen, LDNormal,
  LDTestVal, LDO

MODECLASS Normal
  Initial Standby (~SlfTstPressed500 & ~ResetPressed3000 &
                   ~InTest14000 & ~FlashOff5 & ~ShutdownLockTime200)

  MODE Standby
    Operating @T(ResetPressed3000) WHEN [InsideHysR & ~SlfTstPressed500
                                         & WithinLimits]
    Test @T(SlfTstPressed500)

  MODE Operating
    Shutdown @F(WithinLimits) WHEN [~SlfTstPressed500 & ~InsideHysR]
    Test @T(SlfTstPressed500)

  MODE Shutdown
    Operating @T(InsideHysR) WHEN [~SlfTstPressed500 & WithinLimits
                                  & ~ShutdownLockTime200 ]
    Standby @T(ShutdownLockTime200) WHEN [~SlfTstPressed500]
    Test @T(SlfTstPressed500)

  MODE Test
    Standby @T(InTest14000) WHEN [~SlfTstPressed]
    Standby @F(SlfTstPressed) WHEN [InTest14000]

MODECLASS Failure
  Initial AllOk (~ControlUnitFail & ~TimeDeviceFail & ~LevelDeviceFail)

  MODE AllOkM
    HardFail @T(ControlUnitFail)
    HardFail @T(TimeDeviceFail)
    BadLevDev @T(LevelDeviceFail) WHEN [~ControlUnitFail &
                                        ~TimeDeviceFail]
```

Figure A.2: Requirements for Water-Level Monitoring System.

```
  MODE BadLevDev
    HardFail @T(ControlUnitFail)
    HardFail @T(TimeDeviceFail)

  MODE HardFail

CONTROLLED_VARIABLES
  VARIABLE HighWindowOn
    Initial True
    True @T(LevelHigh) WHEN [Operation=Operating & Failure=AllOk]
    True @T(LevelHigh) WHEN [Operation=Shutdown & Failure=AllOk]
    True @T(InTest0) WHEN [Operation=Test & Failure=AllOk]
    True @T(FlashOff5) WHEN [Failure=BadLevDev]
    False @T(Operation=Operating & Failure=AllOk) WHEN [~LevelHigh]
    False @T(InTest2000) WHEN [Operation=Test & Failure=AllOk]
    False @F(FlashOff5) WHEN [Failure=BadLevDev]

  VARIABLE LowWindowOn
    Initial True
    True @T(LevelLow) WHEN [Operation=Operating & Failure=AllOk]
    True @T(LevelLow) WHEN [Operation=Shutdown & Failure=AllOk]
    True @T(InTest2000) WHEN [Operation=Test & Failure=AllOk]
    True @F(FlashOff5) WHEN [Failure=BadLevDev]
    False @T(Operation=Operating & Failure=AllOk) WHEN [~LevelLow]
    False @T(InTest4000) WHEN [Operation=Test & Failure=AllOk]
    False @T(FlashOff5) WHEN [Failure=BadLevDev]

  VARIABLE LDNormal
    Initial True
    True  @T(Operation=Operating) WHEN [Failure=AllOk]
    True  @T(Operation=Shutdown) WHEN [Failure=AllOk]
    True  @T(Operation=Standby) WHEN [Failure=AllOk]
    False @T(Operation=Test) WHEN [Failure=AllOk]

  VARIABLE LDO
    Initial False
    True @T(InTest0) WHEN [Failure=AllOk]
    False @T(InTest4000) WHEN [Operation=Test & Failure=AllOk]
    True @T(InTest14000) WHEN [Operation=Test & Failure=AllOk]
```

Figure A.2: Requirements for Water-Level Monitoring System (cont'd).

```
  VARIABLE LDTestVal
    Initial False
    False @T(InTest0) WHEN [Failure=AllOk]
    True @T(InTest4000) WHEN [Operation=Test & Failure=AllOk]
    False @T(InTest14000) WHEN [Operation=Test & Failure=AllOk]

  VARIABLE AlarmOn
    Initial False
    True @F(WithinLimits) WHEN [Operation=Operating & Failure=AllOk]
    True @F(WithinLimits) WHEN [Operation=Shutdown & Failure=AllOk]
    True @T(InTest0) WHEN [Operation=Test & Failure=AllOk]
    True @T(Failure=BadLevDev)
    False @T(Operation=Operating & Failure=AllOk) WHEN [WithinLimits]
    False @T(InTest4000) WHEN [Operation=Test & Failure=AllOk]

  VARIABLE PumpSwitchOpen
    Initial False
    True @T(Operation=Standby & Failure=AllOk) WHEN [~PowerFail]
    True @T(Operation=Test & Failure=AllOk) WHEN [~PowerFail]
    True @T(Failure=BadLevDev) WHEN [~PowerFail]
    True @T(Failure=HardFail) WHEN [~PowerFail]
    False @F(ResetPressed) WHEN [~PowerFail]
    False @T(PowerFail)

RELATIONSHIPS
    InsideHysR -> WithinLimits
    SlfTstPressed < SlfTstPressed500
    ResetPressed < ResetPressed3000
    LDTestVal | LD0
    InTest0 < InTest2000 < InTest4000 < InTest14000

PROPERTIES
  wmi(Operation = Operating, ~SlfTstPressed500 & (
      WithinLimits | SlfTstPressed))
  wmi(Operation = Operating, WithinLimits & ~SlfTstPressed500)
  wmi(Operation = Shutdown, (~SlfTstPressed500 &
      ((~InsideHysR & ~ShutdownLockTime200) | ~SlfTstPressed)))
  wmi(Operation = Shutdown, (~InsideHysR & ~ShutdownLockTime200))
  wmi(Operation = Standby, ~SlfTstPressed500)
  cause(@T(SlfTstPressed500), InTest0)
```

Figure A.2: Requirements for Water-Level Monitoring System (cont'd).

```
MONITORED
  Time, In(Manager=Grant1), Done

CONTROLLED
  In(SubSys1=S1Wait), In(SubSys1=S1Done)

MODECLASS SubSys1
  INITIAL S1Done

  MODE S1Done
    S1Wait @T(Time)

  MODE S1Wait
    S1RecGrant @T(In(Manager=Grant1))

  MODE S1RecGrant
    S1Done @T(Done)

CONTROLLED_VARIABLES
  VARIABLE In(SubSys1=S1Wait)
    Initial False
    False @T(In(Manager=Grant1))
    True  @T(Time)

  VARIABLE In(SubSys1=S1Done)
    Initial True
    False @T(Time)
    True @T(Done)
```

Figure A.3: Requirements for SubSys1 modeclass of the MES system.

```
MONITORED
  In(SubSys1=S1Wait), In(SubSys2=S2Wait), In(SubSys1=S1Done),
  In(SubSys2=S2Done)

CONTROLLED
  In(Manager=Grant1), In(Manager=Grant2)

MODECLASS Manager
  INITIAL Stable
  MODE Stable
      Grant1 @T(In(SubSys1=S1Wait))
      Grant2 @T(In(SubSys2=S2Wait))

  MODE Grant1
      Stable @T(In(SubSys1=S1Done)) WHEN [~In(SubSys2=S2Wait)]
      Grant2 @T(In(SubSys1=S1Done)) WHEN [In(SubSys2=S2Wait)]

  MODE Grant2
      Stable @T(In(SubSys2=S2Done)) WHEN [~In(SubSys1=S1Wait)]
      Grant1 @T(In(SubSys2=S2Done)) WHEN [In(SubSys1=S1Wait)]

CONTROLLED_VARIABLES
  VARIABLE In(Manager=Grant1)
      Initial False
      True @T(In(SubSys1=S1Wait)) WHEN [Manager=Stable]
      True @T(In(SubSys2=S2Done)) WHEN [Manager=Grant2 &
                                        In(SubSys1=S1Wait)]
      False @T(In(SubSys1=S1Done))

  VARIABLE In(Manager=Grant2)
      Initial False
      True @T(In(SubSys2=S2Wait)) WHEN [Manager=Stable]
      True @T(In(SubSys1=S1Done)) WHEN [Manager=Grant1 &
                                        In(SubSys2=S2Wait)]
      False @T(In(SubSys2=S2Done))

PROPERTIES
  reach (Manager=Grant1)
  reach (Manager=Grant2)
```

Figure A.4: Requirements for Manager modeclass of MES.

```
COMBINED_PROPERTIES
  strcause (SubSys1=S1Done, Manager=Stable)
  strcause (SubSys2=S2Done, Manager=Stable)
  smi (~(SubSys1=S1RecGrant) | ~(SubSys2=S2RecGrant))
```

Figure A.5: Global properties for MES.

Appendix B

Design Language Constructs

In this chapter, we provide the BNF grammar for the PDL described in Chapters 3 and 6. The language consists of C-like control-flow constructs and annotations. Only annotations are described. Syntax for BNF is reviewed in Appendix A.2.

An annotation is either a sequential or a synchronization construct:

$$\begin{array}{rcl} \textit{annotation} & ::= & \textit{sequent-annot} \\ & || & \textit{synch-annot} \end{array}$$

Sequential annotations are separated into Updates, Initials, and Asserts, and synchronization annotations - into Reads and Writes. Each Update and Initial annotation is followed by a list of variables. Every Assert is followed by a list of ORed expressions.

$$\begin{array}{lcl} \textit{sequent-annot} & ::= & \textbf{UPDATE}\ \textit{cond-list} \\ & || & \textbf{INITIAL}\ \textit{mode-class}\ \textbf{:}\ \textit{mode-name}\ [\ \textbf{,}\ \textit{condition}\ ]* \\ & || & \textbf{ASSERT}\ \textit{cond-list}\ [\ |\ \textit{cond-list}\ ]* \\ \textit{synch-annot} & ::= & \textbf{READ}\ \textit{buffer-info} \\ & || & \textbf{WRITE}\ \textit{buffer-info} \end{array}$$

In the above declaration, *mode-class* and *mode-name* refer to a modeclass and a mode, respectively. Only one Initial annotation is allowed for each modeclass. *cond-list* consists of one or more variables - monitored or controlled, or modeclasses. Top(*variable*) indicates reading in a value of *variable* and corresponds to the maximum, $\top$, value that *variable* can attain.

cond-list	::=	*condition* [& *condition*]*
condition	::=	*variable-expr*
	||	~ *variable-expr*
	||	*variable-expr* = **Top**
	||	**Top** (*variable-expr*)
	||	*variable* = **True**
	||	*variable* = **False**
variable-expr	::=	*mode-variable*
	||	*variable*
mode-variable	::=	*mode-class*(*mode-name*)
	||	*mode-class*=*mode-name*
variable	::=	*id*
	||	**In**(*mode-class*= *mode-name*)

Information on each buffer consists of a label and zero or more variables to be read or written:

buffer-info	::=	*buffer-label* [*variable-list*]*
variable-list	::=	*variable* , *variable-list*
	||	

Appendix C

Case Study: WLMS

In this appendix, we list inputs and outputs of the case-study described in Chapter 5. Requirements specifications for this system are listed in Figure A.2 of Appendix A.2. A portion of the C implementation of WLMS is shown in Figure C.1. We've also included annotations inserted into the code to show how we reverse-engineering the design. Only functions with annotations are inserted. Since our analysis ignores all statements except for annotations and control-flow, this Figure constitutes the analyzed design. Figure C.2 shows portions of output produced by Analyzer during the case-study. Due to the size of output, we have included only error messages and some diagnostic information.

```
...
get_event(){
...
  for (;;) {
   XNextEvent(dpy,&e);
   switch(e.type) {
   case Expose:
     DisplayX();
     break;
   case ButtonPress:
     button_event = (XButtonPressedEvent *) & e;
     if (button_event->window == ResetButton) {
       if (ResetButtonPressed) {
         @@ Assert ResetPressed;
         pressbutton(&ResetButton,&gcReset,black,white);
         @@ Update ~ResetPressed;
         ResetButtonPressed = 0;
       }
       else {
         @@ Assert not ResetPressed;
         pressbutton(&ResetButton,&gcReset,white,black);
         @@ Update ResetPressed;
         ResetButtonPressed = 1;
       }
     }
     else if (button_event->window == TestButton) {
       if (TestButtonPressed) {
         @@ Assert SlfTstPressed;
         pressbutton(&TestButton,&gcTest,black,white);
         @@ Update ~SlfTstPressed;
         TestButtonPressed = 0;
       }
       else {
         @@ Assert ~SlfTstPressed;
         pressbutton(&TestButton,&gcTest,white,black);
         @@ Update SlfTstPressed;
         TestButtonPressed = 1;
       }
     }
```

Figure C.1: WLMS design and implementation.

```
    break;
  ...
}
  ...
void  INIT_LVLF (float LVLCM) {
  ...
 GET_MODE(OperatingClass, OPM);
 GET_MODE(FailureClass, FAM);
 if (MODE_EQ(FAM, AllOK)) {
   @@ Assert Failure(AllOK);
   if (! (MODE_EQ(OPM, Test))) {
     @@ Assert ~Normal(Test);
     @@ Update LDNormal;
     DISPLAY(LVLCM);
   }
   else {
     @@ Assert Normal(Test);
     TEST_TIME(&TSTTIM);
     @@ Update Top(InTest0) and Top(InTest14000);
     /* this sets all InTest variables to Top */
     if ((TSTTIM >= 4000) && (TSTTIM < 14000)) {
       @@ Assert InTest4000 & ~InTest14000;
       TSTVAL = ((TSTTIM - 4000)/1000) * (float)11.1;
       @@ Update LDTestVal;
       DISPLAY(TSTVAL);
     }
     else {
       @@ Assert InTest0 | InTest14000;
       @@ Update LD0;
       DISPLAY(0.0);
      }
    }
  }
}

void  INIT_HWAN (float LVLCM) {
 ...
 GET_MODE(OperatingClass, OPM);
 GET_MODE(FailureClass, FAM);
```

Figure C.1: WLMS design and implementation (cont'd).

```
ALOK = (MODE_EQ(FAM, AllOK));
if ((MODE_EQ(OPM, Operating)) && (ALOK)) {
  @@ Assert Normal(Operating) & Failure(AllOK);
  if (LVLCM >= DPAR_HWL) {
    @@ Assert LevelHigh;
    @@ Update HighWindowOn;
    LIGHT(HighLevel, TRUE);
  }
  else {
    @@ Assert ~LevelHigh;
    @@ Update ~HighWindowOn;
    LIGHT(HighLevel, FALSE);
  }
}
else {
  @@ Assert ~Normal(Operating) | ~Failure(AllOK);
  if ((MODE_EQ(OPM, Shutdown)) && (ALOK)) {
    @@ Assert Normal(Shutdown) & Failure(AllOK);
    if (LVLCM >= DPAR_HWL) {
      @@ Assert LevelHigh;
      @@ Update HighWindowOn;
      LIGHT(HighLevel, TRUE);
    }
    else {
      @@ Assert ~Normal(Shutdown) | ~Failure(AllOK);
      if ((MODE_EQ(OPM, Test)) && (ALOK)) {
        @@ Assert Normal(Test) & Failure(AllOK);
        TEST_TIME(&TSTTIM);
        @@ Update Top(InTest0) and Top(InTest14000);
        if ((TSTTIM >= 0) && (TSTTIM < 2000)) {
          @@ Assert InTest0 & ~InTest2000;
          @@ Update HighWindowOn;
          LIGHT(HighLevel, TRUE);
        }
        else {
          @@ Assert InTest2000;
```

Figure C.1: WLMS design and implementation (cont'd).

```
           @@ Update ~HighWindowOn;
           LIGHT(HighLevel, FALSE);
         }
       }
 ...
}
void  INIT_ALMA (float LVLCM) {
...
 GET_MODE(OperatingClass, OPM);
 GET_MODE(FailureClass, FAM);
 ALOK = (MODE_EQ(FAM, AllOK));
 if ((MODE_EQ(OPM, Operating)) && (ALOK)) {
   @@ Assert Normal(Operating) & Failure(AllOK);
   if ((LVLCM >= DPAR_HWL) || (LVLCM <= DPAR_LWL)) {
     @@ Assert ~WithinLimits;
     @@ Update AlarmOn;
     SOUND;
   }
   else {
     @@ Assert WithinLimits;
     @@ Update ~AlarmOn;
     INIT_ALRM;
   }
 }
 else {
   @@ Assert ~Normal(Operating) | ~Failure(AllOK);
   if ((MODE_EQ(OPM, Shutdown)) && (ALOK)) {
     @@ Assert Normal(Shutdown) & Failure(AllOK);
     if ((LVLCM >= DPAR_HWL) || (LVLCM <= DPAR_LWL)) {
       @@ Assert ~WithinLimits;
       @@ Update AlarmOn;
       SOUND;
     }
     else {
       @@ Assert ~Normal(Shutdown) | ~Failure(AllOK);
       if ((MODE_EQ(OPM, Test)) && (ALOK)) {
         @@ Assert Normal(Test) & Failure(AllOK);
         TEST_TIME(&TSTTIM);
         @@ Update Top(InTest0) and Top(InTest14000);
         if ((TSTTIM >= 0) && (TSTTIM < 4000)) {
           @@ Assert InTest0 & ~InTest4000;
```

Figure C.1: WLMS design and implementation (cont'd).

```
           @@ Update AlarmOn;
           SOUND;
         }
         else {
           @@ Assert InTest4000;
           @@ Update ~AlarmOn;
           INIT_ALRM;
         }
       }
       else {
         @@ Assert ~Normal(Test) | ~Failure(AllOK);
         if (MODE_EQ(FAM, BadLevDev)) {
           @@ Assert Failure(BadLevDevM);
           @@ Update AlarmOn;
           SOUND;
           ...
}

void  INIT_LWAN (float LVLCM) {
...
 GET_MODE(OperatingClass, OPM);
 GET_MODE(FailureClass, FAM);
 ALOK = (MODE_EQ(FAM, AllOK));
 if ((MODE_EQ(OPM, Operating)) && (ALOK)) {
   @@ Assert Normal(Operating) & Failure(AllOK);
   if (LVLCM <= DPAR_LWL) {
     @@ Assert LevelLow;
     @@ Update LowWindowOn;
     LIGHT(LowLevel, TRUE);
   }
   else {
     @@ Assert ~LevelLow;
     @@ Update ~LowWindowOn;
     LIGHT(LowLevel, FALSE);
   }
 }
 else {
   @@ Assert ~Normal(Operating) | ~Failure(AllOK);
   if ((MODE_EQ(OPM, Shutdown)) && (ALOK)) {
     @@ Assert Normal(Shutdown) & Failure(AllOK);
```

Figure C.1: WLMS design and implementation (cont'd).

```
     if (LVLCM <= DPAR_LWL) {
       @@ Assert LevelLow;
       @@ Update LowWindowOn;
       LIGHT(LowLevel, TRUE);
       }
       else {
         if ((MODE_EQ(OPM, Test)) && (ALOK)) {
           @@ Assert Normal(Test) & Failure(AllOK);
           TEST_TIME(&TSTTIM);
           @@ Update Top(InTest0) and Top(InTest14000);
           if ((TSTTIM >= 2000) && (TSTTIM < 4000)) {
             @@ Assert InTest2000 & ~InTest4000;
             @@ Update LowWindowOn;
             LIGHT(LowLevel, TRUE);
           }
           else {
             @@ Assert InTest4000;
             @@ Update ~LowWindowOn;
             LIGHT(LowLevel, FALSE);
           }
         }
         ...
}

void  INIT_TSF() {
...
 GET_MODE(OperatingClass, OPM);
 GET_MODE(FailureClass, FAM);
 ALOK = (MODE_EQ(FAM, AllOK));
 if ((MODE_EQ(OPM, Operating)) && (ALOK)) {
   @@ Assert Normal(Operating) & Failure(AllOK);
   @@ Update ~PumpSwitchOpen;
   START_PUMP;
 }
 else {
   @@ Assert ~Normal(Operating) | ~Failure(AllOK);
   if ((MODE_EQ(OPM, Shutdown)) && (ALOK)) {
     @@ Assert Normal(Shutdown) & Failure(AllOK);
   }
```

Figure C.1: WLMS design and implementation (cont'd).

```
    else {
      @@ Assert ~Normal(Operating) | ~Normal(Shutdown) |
                ~Failure(AllOK);
      @@ Update PumpSwitchOpen;
      STOP_PUMP;
    }
  }
}
 ...
int main() {
  ...
  char* modenames[] = { "AllOK", "BadLevDev", "HardFail", "Operating",
                        "Shutdown", "Standby", "Test" };
  @@ Initial  Normal(StandbyM) & ~SlfTstPressed500 & ~ResetPressed3000
    & ~InTest14000 & ~FlashOff5 & ~ShutdownLockTime200 & HighWindowOn
    & LowWindowOn & LDNormal & ~LD0 & ~LDTESTVAL & ~PumpSwitchOpen &
    ~PowerFail & Failure(AllOK) & ~ControlUnitFail &  ~TimeDeviceFail
    & ~LevelDeviceFail;
  ...
  SET_MODE(OperatingClass, Standby);
  SET_MODE(FailureClass, AllOK);

  while (TRUE) {
    get_event();
    GET_TIME(BEEPER, &BEEPTM);
    @@ Update Top(FlashOff5);
    ...
    GET_MODE(FailureClass, FACM);
    if (MODE_EQ(FACM, BadLevDev)) {
      @@ Assert Failure(BadLevDevM);
      if (BEEPTM / 500 % 2 == 0) {
        @@ Assert FlashOff5;
        LIGHT(HighLevel, TRUE);
        LIGHT(LowLevel, FALSE);
        @@ Update HighWindowOn & ~LowWindowOn;
      } else {
        @@ Assert ~FlashOff5;
        LIGHT(HighLevel, FALSE);
        LIGHT(LowLevel, TRUE);
        @@ Update ~HighWindowOn & LowWindowOn;
      }
    }
```

Figure C.1: WLMS design and implementation (cont'd).

```
...
GET_MODE(OperatingClass, OPCM);
WATER_LEVEL(&LVLCM);
INLIM = ((LVLCM < (float)DPAR_HWL) && (LVLCM > (float)DPAR_LWL));
@@ Update Top(WithinLimits);
if (PRESSED(Reset) && INLIM && MODE_EQ(OPCM, Standby)) {
  @@ Assert Normal(StandbyM);
  @@ Assert WithinLimits;
  @@ Update Normal(Operating);
  SET_MODE(OperatingClass, Operating);
}

GET_MODE(OperatingClass, OPCM);
WATER_LEVEL(&LVLCM);
INLIM = ((LVLCM < (float)DPAR_HWL) && (LVLCM > (float)DPAR_LWL));
@@ Update Top(WithinLimits);
GET_TIME (SHUTER, &SHUTTM);
@@ Update Top(ShutdownLockTime200);
...
if (MODE_EQ(OPCM, Shutdown) && INLIM && (SHUTTM < DPAR_SLT)
    && !(PRESSED(SlfTst))) {
  @@ Assert Normal(Shutdown) & WithinLimits & ~SlfTstPressed
     & ~ShutdownLockTime200;
  @@ Update Normal(Operating);
  SET_MODE(OperatingClass, Operating);
}
...
GET_MODE(OperatingClass, OPCM);
GET_TIME(SHUTER, &SHUTTM);
@@ Update Top(ShutdownLockTime200);
if ((MODE_EQ(OPCM, Shutdown)) && (SHUTTM >= DPAR_SLT)
    && (!(PRESSED(SlfTst)))) {
  @@ Assert Normal(Shutdown) & ~SlfTstPressed
     & ~ShutdownLockTime200;
  @@ Update Normal(StandbyM);
  SET_MODE(OperatingClass, Standby);
}
...
GET_MODE(OperatingClass, OPCM);
TEST_TIME(&TSTTIM);
@@ Update Top(InTest0) & Top(InTest14000);
```

Figure C.1: WLMS design and implementation (cont'd).

```
if ((MODE_EQ(OPCM, Test)) && (TSTTIM >= 14000)) {
  @@ Assert Normal(Test);
  @@ Assert InTest14000;
  @@ Update Normal(StandbyM);
  SET_MODE(OperatingClass, Standby);
}
...
GET_MODE(OperatingClass, OPCM);
if ((!(MODE_EQ(OPCM, Test))) && (PRESSED(SlfTst))) {
  @@ Assert Normal(Operating) & Normal(Shutdown) &
     Normal(StandbyM) & SlfTstPressed;
  @@ Update Normal(Test);
  SET_MODE(OperatingClass, Test);
  INIT_DRVS();
}
...
GET_MODE(OperatingClass, OPCM);
WATER_LEVEL(&LVLCM);
@@ Update Top(WithinLimits);
INLIM = ((LVLCM < (float)DPAR_HWL) && (LVLCM > (float)DPAR_LWL));
if ((MODE_EQ(OPCM, Operating)) && (!(INLIM)) &&
  (! (PRESSED(SlfTst)))) {
  @@ Assert Normal(Operating) & ~WithinLimits & ~SlfTstPressed;
  @@ Update Normal(Shutdown);
  SET_MODE(OperatingClass, Shutdown);
  ...
}
WATER_LEVEL(&LVLCM);
@@ Update Top(LevelHigh) & Top(LevelLow) & Top(WithinLimits);
INIT_LVLF(LVLCM);
INIT_HWAN(LVLCM);
INIT_LWAN(LVLCM);
INIT_ALMA(LVLCM);
INIT_TSF();
...
if (KEYVAL == 81) {
  @@ Update PowerFail;
  break;
}
```

Figure C.1: WLMS design and implementation (cont'd).

```
    if (KEYVAL == 108) {
      @@ Update LevelDeviceFail;
      NOLVDV = 1;
    }
    if (KEYVAL == 119) {
      @@ Update TimeDeviceFail;
      NOWDOG = 1;
    }
    GET_MODE(FailureClass, FACM);
    if ((MODE_EQ(FACM, AllOK)) && (NOLVDV == 1) && (NOWDOG == 0)) {
      @@ Assert Failure(AllOK) & LevelDeviceFail & ~TimeDeviceFail;
      @@ Update Failure(BadLevDevM);
      SET_MODE(FailureClass, BadLevDev);
    }
  }
  @@ Update PumpSwitchOpen;
  STOP_PUMP;
  ...
}
```

Figure C.1: WLMS design and implementation (cont'd).

```
               ANALYZER (Version 3.00)

         **** UNREACHABLE NODES: ****

       on line  946 of function INIT_HWAN
       on line  949 of function INIT_HWAN
       on line  954 of function INIT_HWAN
       on line 1068 of function INIT_LWAN
       on line 1071 of function INIT_LWAN
       on line 1076 of function INIT_LWAN
       on line 1004 of function INIT_ALMA
       on line 1007 of function INIT_ALMA
       on line 1012 of function INIT_ALMA
       on line 1020 of function INIT_ALMA

        **** Verifying Automatic Properties ****

*******  Line 1179 *********
Found an event on line 1164
Found rule:
 HighWindowOn  @T(FlashOff5) WHEN [Failure(BadLevDev)]      TRUE
Found rule:
 LowWindowOn   @T(FlashOff5) WHEN [Failure(BadLevDev)]      FALSE
Error on line 1179:
No event found for transition for HighWindowOn FALSE -> TRUE
No event found for transition for LowWindowOn  TRUE  -> FALSE

*******  Line 1184 *********
Found an event on line 1164
Found rule:
 HighWindowOn  @F(FlashOff5) WHEN [Failure(BadLevDev)]      FALSE
Found rule:
 LowWindowOn   @F(FlashOff5) WHEN [Failure(BadLevDev)]      TRUE
Error on line 1184:
No event found for transition for HighWindowOn FALSE -> TRUE
No event found for transition for LowWindowOn  TRUE  -> FALSE
```

Figure C.2: Results of WLMS analysis.

```
*******  Line 1198 *********
Found an event on line 1194
Error on line 1198 of function main, in modeclass Normal:
        no transition from mode(s) {Standby} to mode(s) {Operating}.

*******  Line 1215 *********
Found an event on line 1207
Error on line 1215 of function main, in modeclass Normal:
        no transition from mode(s) {Shutdown} to mode(s) {Operating}.

*******  Line 1227 *********
Found an event on line 1223
Error on line 1227 of function main, in modeclass Normal:
        no transition from mode(s) {Test} to mode(s) {Standby}.

*******  Line 1239 *********
Found an event on line 1235
Error on line 1239 of function main, in modeclass Normal:
        no transition from mode(s) {Standby} to mode(s) {Test}.

*******  Line 1247 *********
Found an event on line 1235
Error on line 1247 of function main, in modeclass Normal:
        no transition from mode(s) {Standby} to mode(s) {Test}.
Error on line 1247:
No event found for transition for mode Normal Standby -> Test
Total number of errors:
        2 for mode transitions

*******  Line 1259 *********
Found an event on line 1255
Error on line 1259 of function main, in modeclass Normal:
        no transition from mode(s) {Operating} to mode(s) {Shutdown}.

Total number of errors:
        2 for mode transitions

*******  Line 883 *********
Variable LDNormal remains TRUE
```

Figure C.2: Results of WLMS analysis (cont'd).

```
*******  Line 893 *********
Found an event on line  889
Found rule:
 LDTestVal  @T(InTest4000) WHEN [(Normal(Test) & Failure(AllOK))]  TRUE
Found rule:
 LDO        @T(InTest4000) WHEN [(Normal(Test) & Failure(AllOK))]  FALSE

Error on line 893:
No event found for transition for LDTestVal FALSE -> TRUE
No event found for transition for LDO TRUE -> FALSE

*******  Line 898 *********
Found an event on line  889
Found rule:
 LDO        @T(InTest14000) WHEN [(Normal(Test) & Failure(AllOK))] TRUE
Found rule:
 LDO        @T(InTest0) WHEN [Failure(AllOK)]                      TRUE
Error on line 898:
No event found for transition for LDO FALSE -> TRUE

*******  Line 923 *********
Found an event on line 889
Error on line 923 of function INIT_HWAN in variable HighWindowOn:
        no transition from value FALSE to value TRUE.

Found an event on line 1266
Found rule for transition on line 923:
 HighWindowOn  @T(LevelHigh) WHEN [(Normal(Operating)
               & Failure(AllOK))]                         TRUE
Error on line 923:
No event found for transition for HighWindowOn FALSE -> TRUE

*******  Line 928 *********
Found an event on line 889
Found an event on line 1266
Error on line 928 of function INIT_HWAN in variable HighWindowOn:
        no transition from value TRUE to value FALSE.

Total number of errors
        2 for controlled variables
```

Figure C.2: Results of WLMS analysis (cont'd).

```
*******  Line 938 *********
Found an event on line  889
Error on line 938 of function INIT_HWAN, in variable HighWindowOn:
        no transition from value FALSE to value TRUE.

Found an event on line 1266
Found rule:
 HighWindowOn   @T(LevelHigh) WHEN [(Normal(Shutdown) &
                Failure(AllOK))]                           TRUE
Error one line 938:
No event found for transition for HighWindowOn FALSE -> TRUE

*******  Line 981 *********
Found an event on line 1266
Found rule:
 AlarmOn   @F(WithinLimits)
           WHEN [(Normal(Operating) & Failure(AllOK))]      TRUE
Error on line 981:
No event found for transition for AlarmOn FALSE -> TRUE

Found an event on line 1266
Error on line 981 of function INIT_ALMA, in variable AlarmOn:
        no transition from value FALSE to value TRUE.

*******  Line 986 *********
Found an event on line 1266
Error on line 986 of function INIT_ALMA, in variable AlarmOn:
        no transition from value TRUE to value FALSE.

*******  Line 996 *********
Found an event on line 1266
Found rule:
 AlarmOn   @F(WithinLimits)
           WHEN [(Normal(Shutdown) & Failure(AllOK))]      TRUE
Error on line 996:
No event found for transition for AlarmOn FALSE -> TRUE

Found an event on line 1266
Error on line 996 of function INIT_ALMA, in variable AlarmOn:
        no transition from value FALSE to value TRUE.
```

Figure C.2: Results of WLMS analysis (cont'd).

```
*******  Line 1020 *********
Error on line 1020 of function INIT_ALMA, in variable AlarmOn:
No event found for transition for AlarmOn FALSE -> TRUE

*******  Line 1046 *********
Found an event on line 889

Error on line 1046 of function INIT_LWAN in variable LowWindowOn:
        no transition from value FALSE to value TRUE.
Found an event on line 1266
Found rule:
 LowWindowOn   @T(LevelLow) WHEN [(Normal(Operating)
               & Failure(AllOK)]                          TRUE
Error on line 1046:
No event found for transition for LowWindowOn FALSE -> TRUE

*******  Line 1051 *********
Found an event on line  889
Found an event on line 1246
Error on line 1051 of function INIT_LWAN, in variable LowWindowOn:
        no transition from value TRUE to value FALSE.

Total number of errors:
        2 for controlled variables

*******  Line 1061 *********
Found an event on line  889
Error on line 1061 of function INIT_LWAN, in variable LowWindowOn:
        no transition from value TRUE to value FALSE.

*******  Line 1102 *********
Found an event on line  889
Found an event on line 1266
Error on line 1102 of function INIT_TSF, in variable PumpSwitchOpen:
        no transition from value TRUE to value FALSE.

Total number of errors:
        6 for controlled variables
```

Figure C.2: Results of WLMS analysis (cont'd).

```
Found an event on line  889
Found an event on line 1266
Error on line 1112 of function INIT_TSF, in variable PumpSwitchOpen:
        no transition from value FALSE to value TRUE.
Total number of errors:
        15 for controlled variables

*******  Line 1216 *********
Found an event on line 1216
Found rule:
 LowWindowOn   @T(LevelLow) WHEN [(Normal(Shutdown) &
               Failure(AllOK))]                            TRUE
Error on line 1266:
No event found for transition fro LowWindowOn FALSE -> TRUE

*******  Line 1295 *********
Found an event on line 1288
Found an event on line 1284
Found rule:
 AllOK  @T(LevelDeviceFail) WHEN [((~ControlUnitFail)
        & (~TimeDeviceFail))]                              BadLevDev
Error on line 1295:
No event found for transition for mode Failure AllOK -> BadLevDev
Found an event on line  889
Found an event on line 1266
Error on line 1295 of function main, in modeclass Failure:
        no transition from mode(s) {AllOK} to mode(s) {BadLevDev}.

Total number of errors:
3 for mode transitions

*******  Line 1299 *********
Found an event on line 1266
Found an event on line  889
Found an event on line 1280
Error on line 1299 of function main, in variable PumpSwitchOpen:
        no transition from value FALSE to value TRUE.
```

Figure C.2: Results of WLMS analysis (cont'd).

```
                *** Unused Mode Transitions ***
 FROM                          EVENT                                  TO
-------------------------------------------------------------------------
BadLevDev     @T(TimeDeviceFail)                                 HardFail
BadLevDev     @T(ControlUnitFail)                                HardFail
AllOK         @T(TimeDeviceFail)                                 HardFail
AllOK         @T(ControlUnitFail)                                HardFail
Shutdown      @T(SlfTstPressed500)                               Test
Shutdown      @T(ShutdownLockTime200) WHEN [(~SlfTstPressed)]    Standby
Shutdown      @T(InsideHysR) WHEN [(((~SlfTstPressed500) &
                (~ShutdownLockTime200)) & WithinLimits)]         Operating
Test          @F(SlfTstPressed) WHEN [(InTest14000)]             Standby
Test          @F(InTest14000) WHEN [(~SlfTstPressed)]
Operating     @T(SlfTstPressed500)                               Test
Operating     @F(WithinLimits) WHEN [((~SlfTstPressed500) &
                (~InsideHysR))]                                  Shutdown
Standby       @T(SlfTstPressed500)                               Test
Standby       @T(ResetPressed3000) WHEN [((InsideHysR &
                (~SlfTstPressed500)) & WithinLimits)]            Operating


  VARIABLE                          EVENT                           VALUE
---------------------------------------------------------------------
AlarmOn         @T(Failure(BadLevDev))                           TRUE
AlarmOn         @T(InTest0)
                WHEN [(Normal(Test) & Failure(AllOK))]           TRUE
AlarmOn         @T(InTest4000)
                WHEN [(Normal(Test) & Failure(AllOK))]           FALSE
AlarmOn         @T((Normal(Operating) & Failure(AllOK)))
                WHEN [WithinLimits]                              FALSE
HighWindowOn    @T(InTest0)
                WHEN [(Normal(Test) & Failure(AllOK))]           TRUE
HighWindowOn    @T(InTest2000)
                WHEN [(Normal(Test) & Failure(AllOK))]           TRUE
HighWindowOn    @T(Normal(Operating) & Failure(AllOK))
                WHEN [~LevelHigh]                                FALSE
LowWindowOn     @T(InTest2000)
                WHEN [(Normal(Test) & Failure(AllOK))]           TRUE
```

Figure C.2: Results of WLMS analysis (cont'd).

```
LowWindowOn      @T(InTest4000)
                 WHEN [(Normal(Test) & Failure(AllOK))]       TRUE
LowWindowOn      @T(Normal(Operating) & Failure(AllOK))
                 WHEN [~LevelLow]                             FALSE
PumpSwitchOpen   @T(Failure(HardFail) WHEN [~PowerFail]       TRUE
PumpSwitchOpen   @T(Failure(BadDevLev) WHEN [~PowerFail]      TRUE
PumpSwitchOpen   @T(Normal(Test) & Failure(AllOK))
                 WHEN [~PowerFail]                            TRUE
PumpSwitchOpen   @T(Normal(StandBy) & Failure(AllOK))
                 WHEN [~PowerFail]                            TRUE
PumpSwitchOpen   @T(PowerFail)                                FALSE
PumpSwitchOpen   @F(ResetPressed) WHEN [~PowerFail]           FALSE
LDNormal         @T(Normal(StandBy) WHEN [Failure(AllOK)]     TRUE
LDNormal         @T(Normal(Shutdown) WHEN [Failure(AllOK)]    TRUE
LDNormal         @T(Normal(Operating) WHEN [Failure(AllOK)]   TRUE
LDNormal         @T(Normal(Test) WHEN [Failure(AllOK)]        TRUE
LDTestVal        @T(InTest14000)
                 WHEN [(Normal(Test) & Failure(AllOK))]       FALSE
LDTestVal        @T(InTest0) WHEN [Failure(AllOK)]            FALSE

 ********  VERIFICATION OF PROPERTIES ************

CAUSE [@T(SlfTstPressed500), InTest0]
 Satisfied by each state in the system!

WMI [Normal(Standby), (~SlfTstPressed500)]
 Satisfied by each state in the system!

WMI [Normal(Shutdown), ((~InsideHysR) & (~ShutdownLockTime200))]
 Satisfied by each state in the system!

WMI [Normal(Shutdown), ((~SlfTstPressed500) & (((~InsideHysR) &
    (~ShutdownLockTime200)) | (~SlfTstPressed)))]
 Satisfied by each state in the system!

WMI [Normal(Operating), (WithinLimits & (~SlfTstPressed500))]
 Error in function INIT_ALMA, line 981

WMI [Normal(Operating), ((~SlfTstPressed500) &
    (WithinLimits | SlfTstPressed))]
Satisfied by each state in the system!
```

Figure C2: Results of WLMS analysis (cont'd).

```
*******  TOTAL NUMBER OF VIOLATIONS: *********

REACH property:                          10 times
OLT properties for modeclasses:          15 times (8 locations)
OLT properties for controlled variables: 37 times (12 locations)
No events found                          15 times
ALT properties for modeclasses:          13 times
ALT properties for controlled variables: 21 times
User-defined properties:                  1 times
```

Figure C2: Results of WLMS analysis (cont'd).

BIBLIOGRAPHY

[ABB+96] Richard J. Anderson, Paul Beame, Steve Burns, William Chan, Francesmary Modugno, David Notkin, and Jon Reese. "Model Checking Large Software Specifications". In *Proceedings of FSE'96 Foundations on Software Engineering*, October 1996. (To appear).

[AFB+88] T. Alspaugh, S. Faulk, K. Britton, R. Parker, D. Parnas, and J. Shore. "Software Requirements for the A-7E Aircraft". Technical report, Naval Research Laboratory, March 1988.

[AG93] J.M. Atlee and J. Gannon. "State-Based Model Checking of Event-Driven System Requirements". *IEEE Transactions on Software Engineering*, pages 22–40, January 1993.

[All83] J.F. Allen. "Maintaining Knowledge about Temporal Intervals". *Communications of ACM*, 26(11), November 1983.

[ASU88] A. Aho, R. Sethi, and J. Ulman. *Compilers: Principles, Techniques, and Tools, Chapter 10.* Addison Wesley, 1988.

[Atl92] J. Atlee. *"Automated Analysis of Software Requirements"*. PhD thesis, University of Maryland, College Park, Maryland, December 1992.

[Bar89] Geoff Barrett. "Formal Methods Applied to a Floating-Point Number System". *IEEE Transactions on Software Engineering*, 15(5):611–621, May 1989.

[Bar95] G. Barrett. "Model Checking in Practice: The t9000 Virtual Channel Processor". *IEEE Transactions on Software Engineering*, 21(2):69–78, February 1995.

[BB87] T. Bolognesi and E. Brinksma. "Introduction to the ISO specification language LOTOS". *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[BCDM86] M.C. Browne, E.M. Clarke, D.L. Dill, and B. Mishra. "Automatic Verification of Sequential Circuits using Temporal Logic". *IEEE Transactions on Computers*, C-35(12):1035–1044, 1986.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.J. Dill, and L.J. Hwang. "Symbolic Model Checking: 10^{20} States and Beyond". In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pages 428–439, June 1990.

[BD91] B. Berthomieu and M. Diaz. "Modeling and Verification of Time Dependent Systems Using Time Petri Nets". *IEEE Transactions on Software Engineering*, 17(3):259–273, March 1991.

[BdS91] F. Boussinot and R. de Simone. "The Estrel Language". *IEEE Transactions on Software Engineering*, 79(9):1293–1304, September 1991.

[BF91] R. Butler and G. Finelli. “The Infeasibility of Experimental Quantification of Life-Critical Software Reliability”. In *Proceedings of the ACM SIGSOFT'91 Conference on Software for Critical Systems*, pages 66–76, 1991.

[BG92] G. Berry and G. Gonthier. “The Estrel Synchronous Programming Language: Design, Semantics, Implementation”. *Science of Computer Programming*, 19(2), 1992.

[BK85] J. Bergstra and J. Klop. “Algebra of Communicating Processes with Abstraction”. *Journal of Theoretical Computer Science*, 37:77–121, 1985.

[BM88] R. Boyer and J. Moore. *A Computational Logic Handbook.* Academic Press, New York, 1988.

[Bro87] Frederick P. Brooks. “No Silver Bullet: Essence and Accidents of Software Engineering”. *IEEE Computer*, pages 10–19, April 1987.

[Bry86] R. E. Bryant. “Graph-based algorithms for boolean function manipulation.”. *Transactions on Computers*, 8(C-35):677–691, 1986.

[Bry92] Randal E. Bryant. “Symbolic Boolean manipulation with ordered binary-decision diagrams”. *Computing Surveys*, 24(3):293–318, September 1992.

[BS93] J. Bowen and V. Stavridou. “Safety-Critical Systems, Formal Methods and Standards”. *Software Engineering Journal*, 8(6):189–209, July 1993.

[CC76] Patrick Cousot and Radhia Cousot. “Static Determination of Dynamic Properties of Programs”. In *Proceedings of the "Colloque sur la Programmation"*, April 1976.

[CC77a] Patrick Cousot and Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model For Static Analysis of Programs by Construction or Approximation of Fixpoints”. In *Proceedings of the 4th POPL*, pages 238–252, Los Angeles, California, 1977.

[CC77b] Patrick Cousot and Radhia Cousot. “Static Determination of Dynamic Properties of Generalized Type Unions”. *SIGPLAN Notices*, 12(3), March 1977.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications”. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CG75] S.H. Caine and E.K. Gordon. “PDL: A Tool for Software Design”. In *Proceedings of the National Computer Conference*, volume 44, pages 271–276, 1975.

[CG87] E.M. Clarke and O. Grumberg. “Avoiding The State Explosion Problem in Temporal Logic Model Checking Algorithms”. In *??*, pages 293–303, 1987.

[CG94] M. Chechik and J. Gannon. "Automatic Verification of Requirements Implementations". In *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)*, pages 1–14, Seattle, Washington, August 1994.

[CG95] M. Chechik and J. Gannon. "Automatic Analysis of Consistency Between Implementations and Requirements: A Case Study". In *Proceedings of 10th Annual Conference on Computer Assurance*, pages 123–131, June 1995.

[CGH+93] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. Long, K.L. McMillan, and L.A. Ness. "Verification of the Futurebus+ Cache Coherence Protocol". In *Proceedings of CHDL*, 1993.

[CGL92] Edmind M. Clarke, Orna Grumberg, and David E. Long. "Model Checking and Abstraction". In *Proceedings of the Ninth Annual Symposium on Principles of Programming Languages*, pages 343–354, August 1992.

[CGR95] D. Craigen, S. Gerhart, and T. Ralston. "Formal Methods Reality Check: Industrial Usage". *IEEE Transactions on Software Engineering*, 21(2):90–98, February 1995.

[CHK92] Keith D. Cooper, Mary W. Hall, and Ken Kennedy. "Procedure Cloning". In *Proceedings of IEEE International Conference on Computer Languages*, pages 96–105, April 1992.

[CHLR93] P. Clements, C. Heitmeyer, B. Labaw, and A. Rose. "MT: A Toolset for Specifying and Analyzing Real-time Systems". In *Proceedings of Real-Time Systems Symposium*, Raleigh, NC, December 1993.

[CLM89] E. M. Clarke, D. E. Long, and K. L. McMillan. "Compositional Model Checking". In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 464–475, June 1989.

[CM88] K.M. Chandy and J. Mishra. *Parallel Program Design: A Foundation.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.

[CPKM94] A. Coen-Porisini, R. Kemmerer, and D. Mandrioli. "A Formal Framework for ASTRAL Intralevel Proof Obligations". *IEEE Transactions on Software Engineering*, 20(8):548–561, August 1994.

[DC94] Matthew B. Dwyer and Lori A. Clarke. "Data Flow Analysis for Verifying Properties of Concurrent Programs". In *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 62–75, December 1994.

[ea78] John H. Wensley et al. "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control". *Proceedings of IEEE*, 66(10):1240–1255, October 1978.

[EH86] E. Allen Emerson and Joseph Y. Halpern. ""Sometimes" and "Not Never" Revisited: On Branching versus Linear Time Temporal Logic". *Journal of the Association for Computing Machinery*, 33(1):151–178, January 1986.

[Fau89] Stuart Faulk. *"State Determination in Hard-Embedded Systems"*. PhD thesis, University of North Carolina, Chapel Hill, 1989.

[FG94] Jeffrey Fischer and Richard Gerber. "Compositional Model Checking of Ada Tasking Programs". In *Proceedings of the Ninth Annual IEEE Conference on Computer Assurance*, June 1994.

[Flo67] R. Floyd. "Assigning Meanings to Programs". In J. Schwartz, editor, *Symposium on Applied Mathematics: Mathematical Aspects of Computer Science*, volume 19, pages 254–255. American Mathematical Society, New York, 1967.

[FM91] M. Felder and A. Morzenti. "Validating Real-Time Systems by Executing Logic Specifications in TRIO". In *Proceedings of TAV4: the Symposium on testing, Analysis and Verification*, 1991.

[Ger91] R. Gerber. *Communicating Shared Resources: A Model for Distributed Real-Time Systems*. PhD thesis, University of Pennsylvania, 1991.

[GF88] A. Gabrielian and M.K. Franklin. "State-based Specification of Complex Real-time Systems". In *Proceedings of the 9th Real-Time Systems Symposium*, pages 2–11, 1988.

[GH90] G. Guiho and C. Hennebert. "SACEM Software Validation". In *12th International Conference on Software Engineering*, pages 186–191, Nice, France, March 1990.

[GH93a] Hubert Garavel and Rene-Pierre Hautbois. "An Experience with the LOTOS Formal Description Technique on the Flight Warning Computer of the Airbus 330/340 Aircraft". In *First AMAST International Workshop on Real-Time Systems*, Iowa City, IA, November 1993.

[GH93b] J.V. Guttag and J.J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[GHW85] J. Guttag, J. Horning, and J. Wing. "Larch in Five Easy Pieces". Technical Report 5, Systems Research Center, 1985.

[GL91] O. Grumberg and D.E. Long. "Model Checking and Modular Verification". In *Proceedings of CONCUR'91: 2nd International Conference on Concurrency Theory*, 1991.

[GL92] R. Gerber and I. Lee. "A Layered Approach to Automating the Verification of Real-Time Systems". *IEEE Trans. on Software Eng.*, 18(9):768–784, 1992.

[GMM90] Carlo Ghezzi, Dino Mandrioli, and Angelo Morzenti. "TRIO: A Logic Language for Executable Specifications of Real-Time Systems". *Journal of Systems and Software*, 12(2):107–123, May 1990.

[Gor88] M. Gordon. "Mechanizing Programming Logics in Higher Order Logic". Technical Report 145, University of Cambridge, Cambridge, UK, 1988.

[GS93] Dirk Grunwald and Harini Srinivasan. "Data Flow Equations for Explicitly Parallel Programs". In *Proceedings of 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 159–167, May 1993.

[Har87] David Harel. "StateCharts: A Visual Formalism for Complex Systems". *Science of Computer Programming*, 8:231–274, 1987.

[HBGL95] Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruch Labaw. "SCR*: A Toolset for Specifying and Analyzing Requirements". In *Proceedings of 10th Annual Conference on Computer Assurance*, pages 109–122, June 1995.

[Hei96] Constance Heitmeyer. Private communication, 1996.

[Hen80] K. Heninger. "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications". *IEEE Transactions on Software Engineering*, SE-6(1):2–12, January 1980.

[HK91] Iain Houston and Steve King. "CICS Project Report: Experiences and Results from the Use of Z in IBM". In Prehn and Toetenel, editors, *VDM'91: Formal Software Development Methods*, volume 551 of *Lecture Notes in Computer Science*, pages 588–596. Springer-Verlag, October 1991.

[HL85] David Helmbold and David Luckham. "Debugging Ada Tasking Programs". *IEEE Software*, 2(2), March 1985.

[HL93] C. Heitmeyer and B. Labaw. "Consistency Checks for SCR-Style Requirements Specifications". Technical Report NRL Report 93-9586, Naval Research Laboratory, November 1993.

[HL95] M.P.E. Heimdahl and N.G. Levenson. "Completeness and Consistency Analysis of State-Based Requirements". In *Proceedings of the 17th International Conference on Software Engineering*, Seattle, WA, April 1995.

[HLK95] C. Heitmeyer, B. Labaw, and D. Kiskis. "Consistency Checking of SCR-Style Requirements Specifications". In *Proceedings of RE'95 International Symposium of Requirements Engineering*, March 1995.

[HLN+90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. "STATEMATE: A Working Environment for the Development of Complex Reactive Systems". *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[HM83] C. Heitmeyer and J. McLean. "Abstract Requirements Specification: A New Approach and Its Applications". *IEEE Transactions of Software Engineering*, SE-6(6):580–589, September 1983.

[HM96] Constance Heitmeyer and Dino Mandrioli. "Formal Methods for Real-Time Computing: An Overview". In Constance Heitmeyer and Dino Mandrioli, editors, *Formal Methods for Real-Time Computing*. John Wiley & Sons Ltd, 1996.

[Hoa69] C.A.R. Hoare. "An Axiomatic Basis for Computer Programming". *Communications of the ACM*, 12(10):576–580, October 1969.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, New York, 1985.

[How90] W.E. Howden. "Comments Analysis and Programming Errors". *IEEE Transactions on Software Engineering*, 16(1):72–81, January 1990.

[HS95] W.E. Howden and G.M. Shi. "Linear and Structural Event Sequence Analysis". Submitted to ISSTA'96, June 1995.

[HW94] W.E. Howden and B. Wieand. "QDA – A Method for Systematic Informal Program Analysis". *IEEE Transactions on Software Engineering*, 20(6):445–462, June 1994.

[Jac92] D. Jackson. *Aspect: A Formal Specification Language for Detecting Bugs*. PhD thesis, MIT, Cambridge, Massachusetts, June 1992.

[Jac93a] Daniel Jackson. "Abstract Analysis with Aspect". In *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA)*, pages 19–27, June 1993.

[Jac93b] Daniel Jackson. "Aspect: Detecting Bugs with Abstract Dependences". (submitted to Transactions on Software Engineering and Methodology), November 1993.

[Jac94] Daniel Jackson. "Abstract Model Checking of Infinite Specifications". In *Proceedings of FME'94: Industrial Benefit of Formal Methods, Second International Symposium of Formal Methods Europe*, pages 519–531, October 1994.

[JJ91] J.Baeten and J.Bergstra. "Real-Time Process Algebra". *Formal Aspects of Computing*, 3(2):142–188, 1991.

[JLHM91] Matthew S. Jaffe, Nancy G. Levenson, Mats P.E. Heimdahl, and Bonnie E. Melhart. "Software Requirements Analysis for Real-Time Process-Control Systems". *IEEE Transactions on Software Engineering*, 17(3), March 1991.

[JM86] F. Jahanian and A. Mok. "Safety Analysis of Timing Properties in Real-Time Systems". *IEEE Transactions on Software Engineering*, SE-12(9):890–904, September 1986.

[JM87] Farnam Jahanian and Aloysius K.-L. Mok. "A Graph-Theoretic Approach for Timing Analysis and Its Implementation". *IEEE Transactions on Computers*, C-36(8):961–975, August 1987.

[JS88] F. Jahanian and D. Stuart. "A Method for Verifying Properties of Modechart Specifications". In *Proceedings of the Real-Time Systems Symposium*, pages 12–21, 1988.

[Kem85] R.A. Kemmerer. "Testing Formal Specifications to Detect Design Errors". *IEEE Transactions on Software Engineering*, SE-11(1):32–43, January 1985.

[KPS+93] S. Kromodimoeljo, W. Pase, M. Saalttink, D. Craigen, and I. Meisels. "A Tutorial on EVES". Technical report, Odyssey Research Associates, Ottawa, Ontario, 1993.

[Lam88] D.A. Lamb. *Software Engineering: Planning for Change*. Prentice-Hall, 1988.

[LC91] Douglas Long and Lori A. Clarke. "Data Flow Analysis of Concurrent Systems That Use the Rendezvous Model of Synchronization". Technical Report COINS 91-31, Univ. Mass. Amherst, July 1991.

[Lev86] Nancy G. Levenson. "Software Safety: Why, What, and How". *Computing Surveys*, 18(2):125–163, June 1986.

[LRD95] F. Lagnier, P. Raymond, and C. Dubois. "Formal Verification of a Critical System Written in Saga/LUSTRE". In *Workshop on Formal Methods, Modelling and Simulation for System Engineering*, February 1995.

[LS87] Nancy G. Levenson and Janice L. Stolzy. "Safety Analysis Using Petri Nets". *IEEE Transactions on Software Engineering*, SE-13(3):386–397, March 1987.

[LT88] N. Lynch and M. Tuttle. "An Introduction to Input/Output Automata". Technical Report MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology, 1988.

[LV91] N. Lynch and F. Vaandrager. "Forward and backward simulations for timing-based systems". In *Proceedings of the REX Workshop "Real-Time: Theory in Practice"*, LNCS 600. Springer-Verlag, 1991.

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

[Mil89] R. Milner. *Communication and Concurrency*. Prantice-Hall, New York, 1989.

[MLJ88] B.E. Melhart, N.G. Levenson, and M.S. Jaffe. "Analysis Capabilities for Requirements Specified in Statecharts". Technical report, University of California, Irvine, California, 1988.

[MMT91] M. Merritt, F. Modugno, and M. Tuttle. "Time Constrained Automata". In *Proceedings of CONCUR'91: 2nd International Conference on Concurrency Theory*, Amsterdam, The Netherlands, August 1991.

[MP90] Z. Manna and A. Pnueli. "Tools and Rules for the Practicing Verifier". Technical Report STAN-CS-90-1321, Department of Computer Science, Stanford University, 1990.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1992.

[MR91] S.P. Masticola and B.G. Ryder. "A Model for Ada Programs for Static Deadlock Detection in Polynomial Time". *Proceedings of ACM SIGPLAN Programming Languages Design and Implementation, ACM SIGPLAN Notices*, 26(12):97–107, 1991.

[MS76] P.M. Merlin and A. Segall. "Recoverability of Communication Protocols - Implications of a Theoretical Study". *IEEE Transactions on Communications*, pages 1036–1043, 1976.

[MS92] K.L. McMillan and J. Schwalbe. "Formal Verification of the Gigamax Cache Consistency Protocol". In N. Suzuki, editor, *Shared Memory Multiprocessing.* MIT Press, 1992.

[MSS82] P.M. Melliar-Smith and R.L. Schwartz. "Formal Specification and Verification of SIFT: A Fault Tolerant Flight Control System". *IEEE Transactions on Computers*, C-31(7):616–630, July 1982.

[oD84] U.S. Department of Defense. "System Safety Program Requirements". Technical Report MIL-STD-882B, U.S. Government Printing Office, Washington, D.C., March 1984.

[Off92] U.S. General Accounting Office. "Mission Critical Systems: Defense Attempting to Address Major Software Challenges". Technical Report TR 90-276, U.S. General Accounting Office, Washington, DC, December 1992.

[OO89] Kurt M. Olender and Leon J. Osterweil. "Cesar: A Static Sequencing Constraint Analyzer". In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 66–74, December 1989.

[OO90] Kurt M. Olender and Leon J. Osterweil. "Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation". *IEEE Transactions on Software Engineering*, 16(3):268–280, March 1990.

[OO92] Kurt M. Olender and Leon K. Osterweil. "Interprocedural Static Analysis of Sequencing Constraints". *ACM Transactions of Software Engineering and Methodology*, 1(1):21–52, January 1992.

[OSR93] S. Owre, N. Shankar, and J. Rushby. "User Guide for the PVS Specification and Verification System (Draft)". Technical report, Computer Science Lab, SRI International, Menlo Park, CA, 1993.

[Ost89a] J. Ostroff. "Synthesis of Controllers for Real-time Discrete Event Systems". In *Proceedings of the 28th IEEE Conference on Decision and Control*, 1989.

[Ost89b] J. Ostroff. *Temporal Logic For Real-Time Systems.* Research Studies Press LTD, Taunton, Somerset, England, 1989.

[Ost91] J. Ostroff. "Systematic Development of Real-time Discrete Event Systems". In *Proceedings of the ECC91 European Control Conference*, pages 522–533, Paris, 1991.

[Par85] David Lorge Parnas. "Software Aspects of Strategic Defense Systems". *American Scientist*, 73:432–440, September-October 1985.

[Par93] D.L. Parnas. "Some Theorems We Should Prove". In *Proceedings of 1993 International Conference on HOL Theorem Proving and Its Applications*, Vancouver, BC, August 1993.

[Per87] Dewayne E. Perry. "Software Interconnection Models". In *Proceedings of the 9th International Conference on Software Engineering*, pages 61–69. IEEE Computer Society Press, 1987.

[Per89a] Dewayne E. Perry. "The Inscape Environment.". In *Proceedings of the 11th International Conference on Software Engineering*, pages 60–68, Pittsburgh PA, May 1989.

[Per89b] Dewayne E. Perry. "The Logic of Propagation in The Inscape Environment". In *Proceedings of the 3rd Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 114–121, Key West, Florida, December 1989.

[Pet81] J. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, 1981.

[PGJI93] P.Bremond-Gregoire, J.Y.Choi, and I.Lee. "The Soundness and Completeness of ACSR". Technical Report MS-CIS-93-59, University of Pennsylvania, June 1993.

[PM91] D. Parnas and J. Madey. "Functional Documentation for Computer Systems Engineering(Version 2)". Technical Report CRL Report 237, McMaster University, Department of Electrical and Computer Engineering, 1991.

[PM95] David Lorge Parnas and Jan Madey. "Functional Documents for Computer Systems". *Science of Computer Programming*, 25:41–61, 1995.

[Pnu77] A. Pnueli. "The Temporal Logic of Programs". In *Proceedings of 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57, 1977.

[PP94] D. Peters and D.L. Parnas. "Generating a Test Oracle from Program Documentation". In *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)*, pages 58–65, Seattle, WA, August 1994.

[PW94] D.L. Parnas and Y. Wang. "Simulating the Behaviour of Software Modules by Trace Rewriting Systems". *IEEE Transactions on Software Engineering*, 20(10):750–759, 1994.

[QS81] J.P. Quielle and J. Sifakis. "Specification and Verification of Concurrent Systems in CESAR". In *Proceedings of the Fifth International Symposium in Programming*, 1981.

[Ram74] C. Ramchandani. "Analysis of Asynchronous Concurrent Systems by Timed Petri Nets". Technical report, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1974.

[Rei85] W. Reisig. *Petri Nets: An Introduction.* Springer-Verlag, Berlin, 1985.

[RPRL96] Vivek Ratan, Kurt Partridge, Jon Reese, and Nancy Levenson. "Safety Analysis Tools for Requirements Specifications". In *Proceedings of COMPASS'96*, Gathersburg, Maryland, June 1996. (to appear).

[RR87] G. Reed and A. Roscoe. "Metric Spaces as Models for Real-Time Concurrency". In *Proceedings of Mathematical Foundations of Computer Science, LNCS*, volume 298, New York, 1987. Springer-Verlag.

[Rus93] John Rushby. "Formal Methods and the Certification of Critical Systems". Technical Report CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.

[SA96] Tirumale Sreemani and Joanne M. Atlee. "Feasibility of Model Checking Software Requirements: A Case Study". In *Proceedings of COMPASS'96*, Gathersburg, Maryland, June 1996.

[SMS82] R.L. Schwartz and P.M. Melliar-Smith. "From State Machines to Temporal Logic: Specification Methods for Protocol Standards". *IEEE Transactions on Communications*, Com-30, 1982.

[Tay83] Richard N. Taylor. "A General-Purpose Algorithm for Analyzing Concurrent Programs". *Communications of the ACM*, 26(5):362–376, May 1983.

[Vai91] C. Vail. *"Program Verification via Abstraction using Incremental Operational Specifications"*. PhD thesis, University of California, San Diego, 1991.

[van90] A. J. van Schouwen. "The A-7 Requirements Model: Re-examination for Real-Time Systems and an Application to Monitoring Systems". Technical Report TR-90-276, Queen's University, Kingston, Ontario, May 1990.

[vPM93] A.J. vanShouwen, D.L. Parnas, and J. Madey. "Documentation of Requirements for Computer Systems". In *Proceedings of RE'93 International Symposium on Requirements Engineering*, San Diego, January 1993.

[Weg75] Ben Wegbreit. "Property Extraction in Well-Founded Property Sets". *IEEE Transactions on Software Engineering*, 1(3):270–285, September 1975.

[WVF95] Jeannette Wing and Mandana Vaziri-Farahani. "Model Checking Software Systems: A Case Study". In *Proceedings of the 3rd Symposium on the Foundations of Software Engineering*, pages 128–139, October 1995.

[WZ91] Mark N. Wegman and Kenneth Zadeck. "Constant Propagation". In Fran Allen, Barry Rosen, and Kenneth Zadeck, editors, *Optimization in Compilers*. ACM Press, 1991. (forthcoming).

[YMW93] J. Yang, A.K. Mok, and F. Wang. "Symbolic Model Checking for Event-Driven Real-Time Systems". In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, Raleigh-Durham, NC, December 1993.