Implementing a Hazard Elimination Analysis Tool for SpecTRM-RL Using Backwards Reachability

by

Kenneth K. Lu

S.B., Massachusetts Institute of Technology (2000)

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in
Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2002

Author ....[illegible]........
Department of
Electrical Engineering and Computer Science
May 28, 2002

Certified by ....[illegible]........
Nancy G. Leveson
Professor
Thesis Supervisor

Accepted by ....[illegible]........
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Implementing a Hazard Elimination Analysis Tool for SpecTRM-RL Using Backwards Reachability

by

Kenneth K. Lu

Submitted to the Department of
Electrical Engineering and Computer Science
on May 28, 2002, in partial fulfillment of the
requirements for the degree of
Master of Engineering in
Electrical Engineering and Computer Science

Abstract

Proper specification can improve the safety of a software system. Hazard elimination analysis involves marking certain system states in a specification as "hazardous", searching for "critical states" which can lead to such hazards, and redesigning the system to avoid the hazardous states. "Backwards reachability" is a technique that can greatly reduce the number of operations required to perform this analysis.

This project implements a hazard elimination analysis algorithm that directly operates on the data files of SpecTRM, a commercial specification environment. The algorithm derives total system states from the various components of a SpecTRM model and determines reachability between any two states. It then applies the Hazard Automaton Reduction Algorithm using backwards reachability to determine the critical states associated with any hazardous state in a given model. The implemented algorithm demonstrates that the technique is effective and efficient.

Thesis Supervisor: Nancy G. Leveson
Title: Professor

Acknowledgments

A deluge of thanks to Prof. Nancy Leveson for her patience and for trusting me to complete my thesis. A ludicrous amount of thanks to Natasha Neogi for her constant encouragement and support. I couldn't possibly have asked for a better mentor to work with. When I doubted myself, she just urged me on nonjudgmentally, which was exactly what I needed. Without her, I would still be running in circles like a headless chicken. A plethora of thanks to Jeffrey Howard over at Safeware Engineering for his unbelievably quick replies to all my emailed cries for help. "User support is part of my job," he'd say, after saving me hours of anguish by replying to a half dozen emails.

A ton of distributed thanks to all the fine folks on the "help class" on MIT Zephyr who are the very models of Good Samaritans.

An infinite amount of thanks to my parents, who I can rely on for understanding and support.

And to my friends... A hop-hop of thanks to Willa AuYeung for always having the knack of calming me down. Two degrees of thanks to Jesse Byler, who gave me company in non-graduating misery and will soon give me company in graduating delight. Many kilograms, kilometers, and kiloliters of thanks to Matt Deeds, who helped me put my thesis on track by giving me honest criticism as I started to write it, and who proofread my thesis after I wrote it. Many long early-morning hours of thanks to Paul Lujan, my oldest friend, who's responsible for keeping me sane with his link-swapping company as I worked all night in these final weeks and also for proofreading and advising fun. Sknaht to Tania Tam for being me. Without my constant and unwavering phone support, I don't know *where* she'd be today. She certainly wouldn't be finishing her thesis, that's for sure! Gracias to Jenny Wang for making the struggle of dealing with the Panamanian phone system worth the effort.

Finally, thanks and thanks again to all my friends at MIT this year for reminding me that this place can be enjoyable, too!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The study of software safety is relatively new compared to other fields of engineering. Poorly-written software specifications have contributed to a number of spacecraft and other accidents. In software systems structured as state machines, "hazard elimination analysis" can improve system safety. This type of analysis involves marking certain system states as "hazardous", searching for "critical states" which can lead to such hazards, and redesigning the system to avoid the hazardous states. Unfortunately, mapping out all possible execution paths of a system can be very time-consuming in a large system. Natasha Neogi has described a more efficient method of searching for critical states using "backwards reachability". [3] This project implements her algorithm and demonstrates its efficiency.

1.1 Motivation

Safety is a important concern when designing the complex systems found in such applications as aircraft, missiles, and nuclear power plants. Human lives often depend on the proper functioning of such systems. Hardware safety is fortunately well-understood. Engineers can calculate overall probabilities of component failure to great accuracy and precision, thus allowing decision makers to balance acceptable risk and cost. Software safety, however, is a younger field. It differs from hardware safety in key ways, both positive and negative.

Without the need for machining, a software package can be produced after a design change in, minutes or hours, not days or months. Unfortunately, this capability for fast feedback is often abused; software engineers tend to design their products by trial and error without careful consideration. The SpecTRM requirements specifications system is designed to encourage more deliberate design by easing the burden of writing formal specifications. [2]

Even in engineering environments where specifications are written, most testing tends to focus on ensuring that the software product conforms to those specifications; little effort is dedicated to ensuring that the specifications are complete and correct to begin with. Fortunately, because software does not generally deteriorate over time by a probabilistic process, the consequences of various system states can be predicted, and a specification can be analyzed for hazards.[3]

1.2 Goal and Approach

"Hazard elimination analysis" applies to systems that can be modeled as state machines. A "hazardous state" is a state of the system which leads to injury or losses. "Critical states" are states which may or may not lead to hazardous states. Analysis consists of the following three steps:

1. The user provides a system model and list of known hazardous states.
2. The analysis finds all derived hazardous states and all critical states.
3. For each hazardous state, the analysis provides the nearest critical state if one exists, allowing the system designer to find a way around the hazard.

1.2.1 Direction of Searching

A forward search from all possible start states would accomplish the analysis. Unfortunately, whether the search is breadth-first or depth-first, a large percentage of the total state space must be searched before all hazardous states are likely to be found.

Worse still, if a hazardous state is actually unreachable, the entire reachable state space must be searched.

A backward search would start from the known hazardous states and recursively check their predecessors until it finds critical states or dead-ends. Thus, it would search through the minimum number of states required. This is the technique that this project explores.

1.2.2 Environment

The SpecTRM environment is designed to be readable by both humans and machines. The goal of this project is to implement a hazard elimination tool using backward reachability which operates on SpecTRM data files, thus demonstrating the feasibility of implementing automated hazard elimination analysis in a real-world product.

1.3 Overview

Chapter 2 explains the background of the SpecTRM environment. Chapters 3 through 6 describe the design of the various components of this tool. Chapter 7 provides examples of operation. Finally, Chapter 8 reveals some additional features that should be implemented in the future.

Chapter 2

The SpecTRM Environment

The tool written for this project operates on data files created in the SpecTRM environment. SpecTRM allows users to write formal specifications in a unique documentation format. Appendix A contains the specification of a sample SpecTRM model used in Chapter 7. The following sections will refer to portions of this model specification and give a brief overview of relevant components. The SpecTRM User Manual [4] contains a more detailed description of the SpecTRM environment.

2.1 Basic Structure

- "Device" elements send and receive signals to the system. They represent the outside world.
- "Input" elements are interfaces between devices and the system. They describe when to pass device inputs on to the rest of the system, and they do not rely on the internal state of the system.
- "Output" elements specify the conditions under which signals are sent to devices.
- "Messages" are the mechanism through which information is passed from devices to inputs and from outputs to devices.

- "Mode" elements describe the mode that the system is in. They can only take on enumerated constants as positions. "Condition tables" specify the triggering conditions for these position.
- "State" elements (or "State Values") represent portions of the internal state of the system. They can also only take on enumerated constants as positions.
- "Macro" elements abstract portions of condition tables out to manage the complexity of large systems.
- "Power Up" is not an explicit model element, though it may be considered a singleton element. It is a global value which is true only during the very first step of system startup.

The names of all model elements and all mode positions reside in the same name space and must all be unique.

The hazard elimination analysis tool is only interested in the internal system state. Thus, it is generally not interested in inputs, outputs, messages, and devices.

2.2 Condition Tables and Expressions

Each position in a mode or state element corresponds to a "condition table". Cells in this table can hold one of three values: "T" (true), "F" (false), and "*" (don't care). Each column represents a triggering condition for that position. That is, if *every* value in *any* column of a condition table holds true, the model element in question performs a transition to the associated position.

Behavior of the system is undefined if multiple condition tables in the same model element contain valid columns simultaneously. SpecTRM has consistency and completeness checking tools to prevent such scenarios. The hazard analysis tool assumes that all models have already been verified within SpecTRM, and it cannot guarantee proper performance on incomplete or inconsistent models.

Expressions in condition tables can take one of many forms:

- "Foo in state S" holds true if state element Foo is in position S. The expression refers to the current value of the element by default, but if "Last Value of" is added to the beginning of the expression, it refers to the position the element was in before the most recent transition. Mode element positions can be specified with a similar expression, replacing "in state" with "in mode".

- An identifier by itself can mean one of two things: It can represent a macro, in which case it evaluates to true if the macro evaluates to true, and vice versa, or it can be "syntactic sugaring" for a mode position. For example, "Operational" can be syntactic sugar for "Soda Machine in mode Operational". This syntax is made possible by the requirement that all mode positions be in the same namespace as model element names, so there is no possibility of name conflict.

- "Baz is X" holds true if the input element Baz carries the value X. Note that input elements are generally required to carry a value of "Obsolete" at system start, though this is not always the case. Similar expressions can use inequality operator in the case of numerical input values.

- Expressions can refer to the time when a certain type of message was last sent or received.

- "Power Up" is true if the system is at the first transition. It is false at all other times.

2.3 Execution and Total System State

When the system begins execution, "Power Up" is set to "true", and mode and state elements transition to appropriate positions. Subsequently, they transition to appropriate values based on inputs and internal state, while outputs send messages to devices when appropriate.

At any point in time, the mode elements, the state elements, and the power up variable sit in a specific set of positions. This set of positions is the "total state" of the system. The hazard analysis algorithm operates on these total states; its purpose is to generate portions of a tree that specifies transitions among total states.

Chapter 3

Design Overview

This tool has a modular design that facilitates maintenance and expansion. Each stage of the tool creates a well-defined object as an interface to the next stage. New applications can easily make use of various modules in this tool. In addition, procedural abstraction ensures that incremental changes to the file format in the future may be supported with only minimal modifications to the tool. Figure 3-1 shows the path of execution.

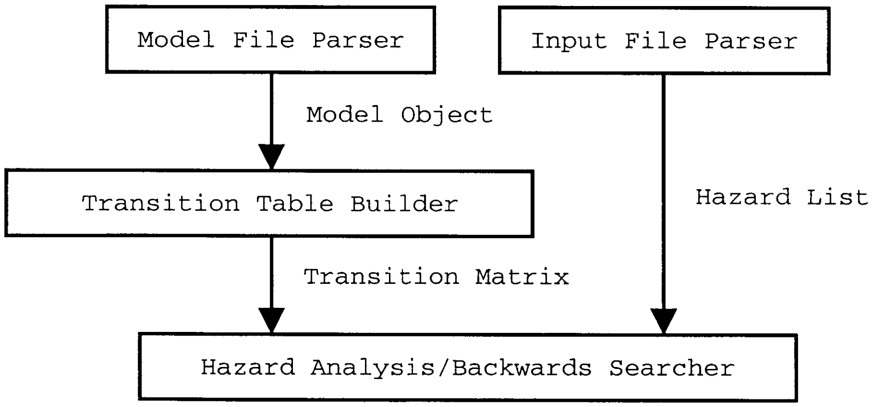

Figure 3-1: Execution Path

1. The user creates a model using SpecTRM. This file must be complete and consistent (as can be confirmed by SpecTRM's built-in model verification tools); the analysis tool does not guarantee proper operation otherwise (with the ex-

ception of omitted “Power Up False” expressions, as discussed in Section 4.5.4).

2. The model file parser reads in the XML data file and produces a model object, translating XML formatting and textual expressions into custom objects. Although these objects are designed for the purposes of reachability analysis, they are also suitable for use by many other types of analysis.

3. The table builder module then uses this model object to create a list of valid “total states” and a two-dimensional matrix of boolean values, mapping destination total states to all the total states which may have been their direct predecessors. Again, both the list of total states (which is tedious to build by hand) and the reachability matrix are very useful for other types of analysis which need to determine which states are possible predecessors or successors of a given state.

4. The input file parser is a simple module which reads in a file containing the initial list of hazardous states and passes them to the hazard analysis module.

5. Finally, the hazard analysis module makes use of the reachability matrix and the input hazard list to perform various searches.

The following chapters will discuss each of these modules in greater detail.

Chapter 4

The Model Object

4.1 Model Class Hierarchy

The model class hierarchy is an abstract representation of the model read from the data file. Its interface is completely file-format-agnostic, thus allowing revised file formats to be supported with no impact to any higher-level functions of the tool. The model class hierarchy is also designed to be as simple as possible, including only those components of the model which are relevant to the problem at hand. If future projects require additional components to be stored, the class can be easily extended to incorporate them. Currently, the hierarchy involves the following:

- A single model object stores its name and an array of its mode and state elements.
- Each model element in turn stores its type, its index among elements of its type, and a list of positions and the rules for transitioning to them.
- Each position stores its type, the index of its associated model element, its index of among positions in that model element, and a list of triggers.
- Each trigger stores a list of expressions derived from SpecTRM-RL. If *every* condition in a trigger holds true, the trigger holds true. If *any* of the triggers associated with a position holds true, a transition to that position should

be made. A trigger loosely corresponds to a column in the truth tables of a SpecTRM model.

In addition, both model elements and positions store references to a master "name lookup table" which maps indices to names.

4.2 XML Parsing

The first step toward analysis is to read the data from the file. SpecTRM data files contain a hierarchical XML structure composed of the model and assorted other specification information. The parser uses the Apache Software Foundation's "Xerces" to parse the subtree containing the model into a Document Object Model (DOM). Various helper functions in the parser are provided for traversing the DOM tree and retrieving key components in accordance with the SpecTRM XML Schema. [5]

4.3 Index-Gathering

Before the bulk of processing, the parser gathers the name and order of all model elements and positions in the model. It creates a number of name-to-index lookup tables. Four tables respectively map names of mode elements, state elements, input elements, and macros to their indices among elements of their type. Two arrays contain, in order, tables which match the positions in each mode and state element to their position indices. Finally, a "reverse mode lookup table" maps mode positions to their respective mode elements. This table is discussed in section 4.4.1.

4.4 Expression Parsing and Trigger Construction

The parser processes each mode and state position's condition table individually. It first uses Étienne Gagnon's "SableCC" to create parse each expression into an abstract syntax tree (AST) based on the SpecTRM-RL grammar. [1] The grammar distinguishes among "Power Ups", mode references, and state references, and the AST reflects that knowledge. The model file parser uses SableCC's tree-walking framework to parse the ASTs into its own custom expression objects which feature more intuitive interfaces and occupy less memory.

4.4.1 Macro References and Sugared Mode Syntax Expansion

SpecTRM-RL employs a "sugared" mode reference syntax where "Operational" can actually mean "Last Value of Soda Machine in mode Operational". Similarly, macro references can simply be written as the names of the macros. In the AST, sugared mode references and macro references are initially indistinguishable; since SableCC reads each expression individually, it stores both as simple identifiers.

The model file parser uses its wider domain of knowledge to differentiate between the two cases. First, it creates a macro name-to-index table and a "reverse mode lookup table" during the "index-gathering pass" that looks up model element and position indices. The reverse mode lookup table maps each mode position name to its associated mode element.

Now, whenever the parser encounters a lone identifier, it looks up the name in the macro and reverse mode lookup tables to determine its type and convert it into the proper custom expression object. (The lookup is possible because mode names must be unique in the same namespace as model element names. [4])

4.4.2 Condition Table to Trigger Conversion

Condition tables, which can require an expression to be false, must be converted to triggers that always require expressions to be true. (See section 4.5.3 for justification of this design decision.) For example:

Foo in state S1	T	*
Bar in state S1	T	*
Bar in state S2	*	T

would be expanded to:

Trigger 1:
Foo in state S1
Bar in state S1

Trigger 2:
Bar in state S2

The absence of any mention to "Foo" in Trigger 2 implies that it can be activated regardless of the position "Foo" is in. Accordingly, if a trigger is empty and contains no condition restrictions, it can always be triggered.

The triggers define *reachability.* Therefore, triggers may overlap. It doesn't matter which of the triggers is activated. As long as *at least one* of the triggers can be activated by a given system state, this position is reachable from that system state. References relevant to the internal state of the system are stored in triggers. References to input values need to be mutually consistent for a given set of triggers to activate, so they are also stored.

Unless the "Relaxed Input Values" flag is set, the parser adds "Power Up is False" as a condition to every trigger which refers to a non-Obsolete input value. (See section 4.5.4 for details.)

"False" Marks in the Condition Table

"False" marks in the condition table present unique challenges for each expression type: "Power Ups" and macros each have only two "positions": true and false. In these cases, "false" marks represent references to the "false" positions of the element in the associated expression, and the parser stores a "Power Up is False" or "Macro Foo is False" expression in the trigger. Expansion becomes more involved in the case of mode and state element references.

Assume that there are two state elements, Foo and Bar. Foo can be in the states S1 through S3, and Bar can be in the states T1 through T4.

Foo in state S1	F

would be expanded to:

Trigger 1:
Foo in state S2

Trigger 2:
Foo in state S3

A more involved example would be:

Foo in state S1	F
Bar in state T1	F
Bar in state T2	F

The parser would expand this in two steps. First, it would expand the "Foo" reference, giving us two triggers with temporary place-holders for the false "Bar" references:

Trigger 1:
Foo in state S2
Bar in state T1 FALSE
Bar in state T2 FALSE

Trigger 2:
Foo in state S3
Bar in state T1 FALSE
Bar in state T2 FALSE

Next, the "Bar" would be expanded. With two restrictions, Bar can only be expanded to "T3" or "T4":

Trigger 1:
Foo in state S2
Bar in state T3

Trigger 2:
Foo in state S2
Bar in state T4

Trigger 3:
Foo in state S3
Bar in state T3

Trigger 4:
Foo in state S3
Bar in state T4

In this way, the parser recursively expands the triggers with "false" references. (References to current and "last" values are treated separately.)

There is one final case to consider:

Foo in state S1	T
Foo in state S2	F

In this case, the parser need not expand the second row because the first row more strictly restricts the values which can trigger the column. In general, the parser only expands "false" references when no "true" references to the same element appear in the same column.

4.5 Design Decisions

4.5.1 Levels of Modularity

After reading in the DOM, the parser must perform complex trigger-expansion operations before it builds the final model. If the parser were to perform the post-processing as it parses the file, trigger-expansion would be integrally tied to the DOM and the SableCC AST, and modularity would be reduced. If it were instead to parse the model into an intermediate representation before performing post-processing, Xerces and SableCC could be replaced with different tools without modifying the trigger-expansion code.

Unfortunately, with this added modularity would come greater complexity. We would need an entirely separate model class hierarchy that provides a static interface while storing the DOMs and ASTs internally. On the other hand, the reduction in modularity is only partial. The expression grammar and XML schema can be changed without affecting the majority of the code; changes to the code in such cases would be limited to the procedurally-abstracted helper functions. I decided that the loss in modularity was unimportant, and that the parser should build the model as it parses the file to avoid unnecessary complexity.

4.5.2 Element And Position Referencing

One of my early design decisions involved component referencing. References to model elements and to positions in those elements are stored by name in the data file, so referencing by name appears to be the easy solution. Referencing by index is also possible, but, because the indices are not directly stored in the file, they would not be available on a single pass through the file; a two-pass parsing process would be required. Nonetheless, the benefits of using indices outweigh the costs:

- The index-gathering pass only needs to traverse to the level of positions and not through all the condition tables.
- Index-gathering and subsequent index lookup each need to run only once, and they do not increase the order of growth of the running time. Referencing by name, on the other hand, would require a name lookup every time a reference is followed. (Even name lookup by hash tables would be slower than array index lookups.)
- Names can still be stored along with indices, so human-readability need not be sacrificed.
- Finally, indices provide a convenient interface for the input of a large number of states during hazard analysis, and they may even be more clear than names to human operators who deal with the same model for an extended period of time.

Thus, the parser makes two passes: One to construct temporary name-to-index lookup tables and one to build the model object, using the lookup tables to store references as indices.

4.5.3 Triggers

The goal of the trigger list abstraction is to simplify model analysis by eliminating conditions that require reference expressions to evaluate as “false”. Such “false” references cannot be allowed when computing unique total system states, and it is thus best to remove this abstraction as soon as possible to improve readability of intermediate data structures.

4.5.4 Power Up Assumptions

It is standard practice within the SpecTRM community to write “Power Up” only when it needs to be true. Condition tables containing the following row are rare:

Power Up	F

Strictly speaking, any column which does not explicitly specify “Power Up” may or may not be a start state. In practice, such columns usually require, directly or indirectly, an input value to be in a position other than “Obsolete”. Since input values must be in “Obsolete” when “Power Up” is true, any other position implies that “Power Up” must be false.

Currently, whenever a column makes a reference to an input value other than “Obsolete”, the parser inserts a “Power Up False” entry into the trigger. If for some reason inputs can be non-Obsolete on startup in a given system, the tool contains a “Relaxed Input Values” flag that prevents automatic insertions of “Power Up False”.

4.5.5 Macros

The parser does not expand macros. Instead, it treats them just like a mode or state element which can only be in one of two states, true or false. (Macros only need to store the contents of one transition table; the “false” position is simply triggered whenever the “true” position isn’t.) Leaving macros as model elements does not actually increase the number of total states because their values are strictly determined by the positions of the remaining elements. On the other hand, expansion of macros would lead to larger and more confusing triggers, which defeats the purposes of macros in the first place.

Chapter 5

The Reachability Table

The reachability table is the centerpiece of the tool. More formally a "state transition matrix", it describes whether one given total state can transition in one step to another given total state. It can thus also provide a list of all total states which are successors or predecessors of a given total state. A variety of searching algorithms can make use of the table, and the table itself will remain backward-compatible as new features are added. The reachability table consists of two elements: a list of all valid total system states and a square boolean matrix.

A "total system state" contains a specified position for every model element, including state elements, mode elements, and the lone "Power Up" element. For example, a total state might be represented as

{Mode Element Foo in mode M1,
State Element Bar in state S1,
State Element Quux in state T1,
Power Up False}

Not all total states are possible. Total states which are internally inconsistent are logically impossible to reach (except in the case of a failure of the state machine itself, in which case all analyses are meaningless anyway). Since no total state may reach nor be reached from an invalid total state, they do not need to be part of the reachability table, and only *valid* total states are included in the "total state list". (Because each element in a total state is mapped to exactly one position, it is safe to refer to "the positions of the total state".)

The square boolean matrix has as each axis a copy of the total state list. Rows of the matrix refer to destinations, and columns refer to sources. If the cell at row i and column j is "true", then total state j may be a predecessor of total state i, and total state i may be a successor of total state j.

5.1 Total State List Construction

The "table builder" routine first recursively iterates through all possible total states and tests them for validity. The number of possible total states is the product of the number of positions in each model element, so it can be quite large. This state space is only sparsely populated with valid states, however, so it is efficient to traverse it once so that invalid states can be quickly noted as such in the future.

5.1.1 Definition of Validity

A total state is "valid" if there exists a set of triggers, one in each position of the total state, which are mutually consistent. Consistency tests fall under two categories:

1. Internal consistency within a trigger. This requires tests against the positions of other elements. "Current Value" expressions that refer to the position of other mode elements, state elements, and the power up status must be consistent with the positions they actually hold in this total state. For instance, assume state element "Foo" has the following trigger for position "S1":

   Trigger 1:
   Bar in state T1

   Any total state containing Foo=S1, Bar=T2 is "invalid" because Bar must always be in state T1 when Foo is in state S1.

   Note that each trigger can be invalidated based on this test independent of any other triggers; only the positions of other elements are relevant for this test.

2. Mutual consistency among triggers of each element. Although "Last Values" cannot be computed when testing the validity of a single total state, requirements among triggers must be consistent. Assume state element "Foo" has the following trigger for position "S1":

   Trigger 1:
   Bar in state T1
   Quux in state V1
   Last Value of Quux in state V1

   Assume also that state element "Bar" has the following trigger for position "T1":

   Trigger 1:
   Quux in state V1
   Last Value of Quux in state V2

   Consider a total state containing Foo=S1, Bar=T1, Quux=V1. Although the triggers for S1 and T1 are both internally consistent in this case, no previous total state could possibly lead to this state. If Quux was in state V1 previously, Bar would not be in T1 now, and if Quux was in V2 previously, Foo would not be in S1 now. Any total state containing Foo=S1, Bar=T1, Quux=V1 is therefore invalid because it is not reachable from any state.

   The table builder must perform similar checks on references to input values. Although input values can be anything at any time, only states which have consistent requirements for input values are valid. This requires comparison not just of discrete values, but also of ranges.

3. Consistency with the Power Up status. When "Power Up" is true, there have been no previous states, so all "Last Value" references are undefined. Consequently, any trigger which requires a specific "Last Value" is automatically disqualified, and any total state which does not satisfy any other triggers in the relevant position is invalid.

5.1.2 Checking Validity

Figure 5-1 details the pseudocode for the recursive algorithm that performs validity checking. The algorithm accepts as input an index to a position in the total state and a list of requirements on "Last Value" and input value references which must hold.

In each loop, the algorithm checks each trigger in the current position for internal consistency and mutual consistency with previous triggers. Once it finds a candidate trigger, it recursively analyzes all positions after the current one to verify the existence of a set of internally consistent triggers in these positions which are mutually consistent with each other and with all triggers found so far.

If the algorithm is able to find a trigger in the current position that can lead to a complete set of consistent triggers, it returns true. Otherwise, it returns false.

This algorithm ensures that these exists a complete set of triggers which are all internally and mutually consistent.

5.2 Matrix Construction

The calculation of the content of each cell, representing whether a source state can transition to a destination state, is independent of the calculation of any other cell. This allows the state transition matrix to act as a cache.

Since backwards searches only read a small number of cells, the matrix only calculates the value of a cell when it is first requested. It then caches the value for future requests. Section 6.6 demonstrates the benefits of this caching or lazy evaluation system.

5.2.1 Definition of Reachability

For the purposes of this tool, a destination total state is "reachable" from a source total state if the source can directly transition to the destination without first transitioning to a third total state.[1]

[1]Generally, a destination total state is considered "reachable" from a source total state if it can be reached after *any number of transitions*. The term is redefined in this tool for lack of a better

A destination is reachable from a source if, in addition to internal consistency, all "Last Value" references in its triggers are consistent with the corresponding positions in the source. For instance, assume that a state in the destination contains the following trigger:

Trigger 1:
Last Value of Bar in state T1

The destination would only be reachable from the source if the source contains the position Bar=T1.

5.2.2 Checking Reachability

The reachability checking algorithm is nearly identical to running the validity checking algorithm on the destination. There are only two additions:

1. Instead of checking for mutual consistency of "Last Value" references among triggers, the algorithm checks for consistency against the actual last values: the positions in the source.

2. If a total state is in "Power Up", it could not have come from any other total state. Thus, if the destination's power up status is true, the algorithm automatically returns false.

adjective to describe the possibility of a one-step transition; its use is clear enough to avoid confusion with the traditional definition.

```
Start with the first position in the total state list and no known
mutual consistency requirements.

Base case: If the algorithm is being called after all positions have been
checked--that is, if the position counter is beyond the number of
elements--we have found a valid set of triggers; return true.

For each trigger in this position:

    For each expression in the trigger:

        Tests 1 & 3: Check for internal consistency of this trigger,
        and check against power up status.

        Test 2: Check against the mutual consistency requirements list
        from previous positions, and take note of any new requirements.

    End For.

    If this trigger is internally consistent:

        Recursively run this algorithm on the next position, taking note of
        the requirements from this position.

        If there exists a set of triggers in the remaining positions which
        are consistent with what we have so far:

            This trigger is good.  No need to check any more triggers in
            this position. Break out of loop.

        Else:

            This trigger isn't good; continue loop and check the next one.

        End If.

    Else if this trigger isn't even internally consistent:

        This trigger isn't good; continue loop and check the next one.

    End If.

End For.

Return true if we've found a good trigger; return false otherwise.
```

Figure 5-1: Pseudocode for validity checker

Chapter 6

The Hazard Elimination Analysis Algorithm

In a safety-critical system, states fall into several categories:

More precisely, these are the three types of states in question:

- Safe states. A safe state is any state which does not necessarily lead to a hazard.
- Hazardous states, or "hazards". A hazard is any state which leads to unacceptable failure (in such forms as damage of equipment or injury to people). If a state causes no problems at the moment but may only lead to hazardous states, it is also hazardous. But if an otherwise safe state has any safe successors, it is not a hazard.
- Unsafe states. These form a superset of hazardous states which have at least a high likelihood of failure. For the purposes of this tool, all unsafe states are considered hazardous.
- Critical states. These form a subset of safe states that have at least one hazardous successor (but also have at least one safe successor).

"Hazard elimination analysis" involves finding the critical states that will lead to each hazardous state. The designer of the system can then reanalyze and redesign the system at those points to prevent or reduce the probability of a transition to a hazardous state, thus increasing the safety of the system.

These are the inputs the user will give to the analysis algorithm:

- A system model. The algorithms discussed in the previous chapters will create a list of valid system states and a (lazily evaluated) transition matrix.
- A list of system states known to be hazardous.

After analysis, the algorithm will provide the user with the following outputs: (There may or may not be actual entries in any given category.)

- A list of input hazards which are invalid.
- A list of input hazards which are valid.
- A list of newly discovered hazards which were not in the input.
- A list of hazard states which are actually start states. This would represent a serious flaw in the design of the system.
- For each hazard, a list of critical states which may eventually lead to that hazard, if any.

This chapter will outline the final algorithm and then justify its design.

6.1 The Final Algorithm

Each hazard has a list of corresponding "sources". At the end of execution, each list of sources will be a list of critical states corresponding to that hazard. During execution, however, sources are merely the closest ancestors of a hazard which are not known hazards. They may be critical states or unanalyzed hazards. The corresponding list of hazards for a source are its "targets".

The algorithm keeps a list of "known hazards" which it expands as it discovers new hazards. It also keeps track of "already processed hazards".

For each valid input hazard, the final algorithm first looks for and handles dead-end cases. (See Section 6.5.)

The algorithm then looks up the predecessors of this hazard:

- If this predecessor is the hazard itself, it is ignored.
- If this predecessor has already been processed, the current hazard "inherits" this predecessor's list of sources. (See Section 6.4.) The algorithm does not add any of these sources that *is* the current hazard, since a hazard cannot be its own source.
- If this predecessor is a known hazard which has not yet been processed, it is added as a source. That known hazard will later update this hazard's source list.
- If this predecessor is not a known hazard, the algorithm checks its successors. If at least one of the successors is not a known hazard, this predecessor is a "possible critical state"; it is simply added to the current hazard's list of sources.
- If all the successors of that predecessor are known hazards, this predecessor is itself a newly discovered hazard; it is added to the list of known hazards as well as to the current hazard's list of sources.

Finally, after analyzing all the predecessors, the algorithm updates the source lists of all targets of the current hazard. It removes the current hazard itself from the source lists, and it adds the current hazard's list of sources to these source lists. Section 6.4.1 will provided a step-by-step sample execution of the algorithm.

6.2 A Basic Algorithm

The high-level approach of a backwards searching algorithm is to check the predecessors of each hazard. If the predecessor is a known safe state, it must be a critical state (since it has a hazardous successor), and the basic algorithm associates that critical state with the hazard and stops execution. If the predecessor is itself a hazard, the algorithm should continue searching until it discovers critical states or a dead end.

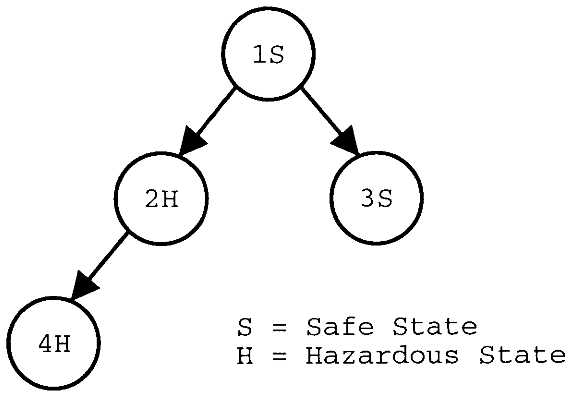

Figure 6-1: Basic State Tree

In Figure 6-1, states 1 and 3 are safe, and states 2 and 4 are hazards. When the basic algorithm searches up from state 2, it sees that state 1 is a critical state, and it associates it with state 2. When the algorithm searches up from state 4, it sees that state 2 is a hazard, so it continues upward until it finds state 1, and it then also associates state 1 with state 4. Neogi describes this basic algorithm in Section 5.4 of her thesis. [3]

Note that the algorithm is performing redundant work by searching state 2's predecessors twice. In fact, when the algorithm finds that state 4's ancestor is also a hazard, it need not continue. It can simply allow state 4 to inherit state 2's critical states.

Unfortunately, there are several factors which complicate this simple approach.

6.3 Detection of Previously Unknown Hazards

The algorithm must be able to detect states that only have hazardous successors, but that the user did not provide as inputs. This means that, whenever a predecessor is checked, the algorithm must check all its successors to see if there are any nonhazards. If there are not, it can label the state a newly-found hazard. Unfortunately, even if there *is* a successor that is not a known hazard, that successor may itself turn out to be a newly-found hazard later on.

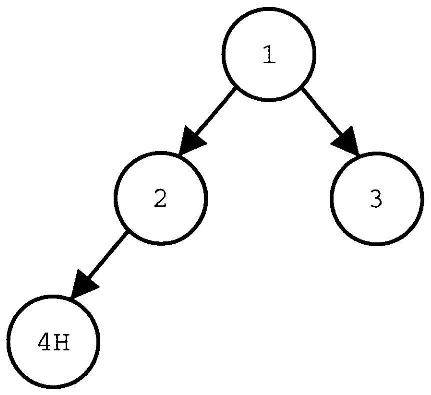

Figure 6-2: State Tree Requiring Hazard Detection

In Figure 6-2, when the algorithm looks at state 4's predecessor, state 2, it sees that it only has hazardous successors (namely, state 4). Thus, the algorithm marks state 2 as a newly discovered hazard.

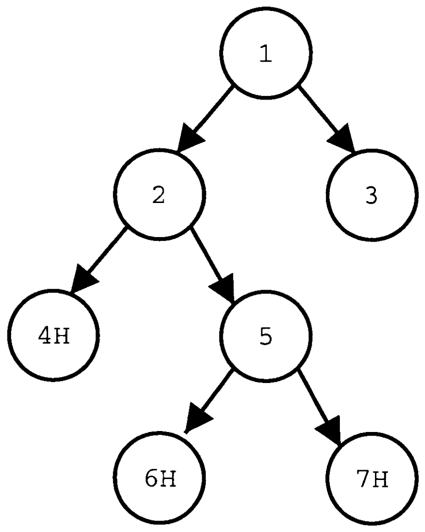

Figure 6-3: State Tree with Delayed Detection

In Figure 6-3, assume that the hazards are analyzed in numerical order. When the algorithm looks at state 4's predecessor, state 2, it seems that it is a critical state.

Unfortunately, state 5 is actually a hazard, and state 2 is thus also actually a hazard.

One obvious solution in this situation would be for the algorithm to continue searching downward whenever a successor is not a known hazard, stopping only when it finds hazards or dead ends. Unfortunately, this could lead to searching through a large number of irrelevant non-hazardous states, which defeats the purpose of using a backwards searching algorithm.

There is a more efficient approach: Simply assume that this predecessor is a critical state for the time being. If its seemingly safe successors turn out to be a hazard later on (state 5 in this case), that state will need to be analyzed. At that point, the algorithm will look at state 5's predecessor, state 2 again, and see that it is now a hazard, marking it for future analysis.

When the algorithm analyzes state 2, it will discover that state 1 is its source, and it can then propagate that source to state 2's children, including state 4. (See section 6.4 for details.)

At the very end of execution, the critical states are any sources that the algorithm never discovered to be hazardous.

6.3.1 Avoiding Hazards Through Loops

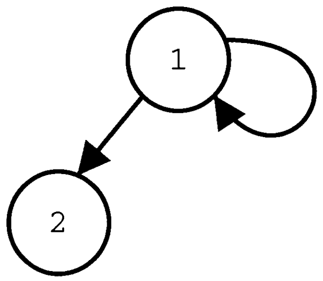

Figure 6-4: State Tree with A Self Loop

Consider Figure 6-4. Although the only apparent successor of state 1 is state 2, state 1 is not a hazard. This is because state 1 is also its *own* successor. Conceptually, one might imagine that a system could simply loop in state 1 forever, never reaching state 2. State 1 *is*, however, a critical state, since it has both hazardous and safe successors.

6.4 Inheritance

The process by which changes are propagated is a key feature of the final algorithm. During analysis, the final algorithm actually associates each hazard with a list of sources which may be either critical states or other hazards which have yet to be analyzed. When a predecessor of a hazard has already been analyzed, the hazard inherits that predecessor's sources.

Conversely, each unanalyzed hazard or possible critical state corresponds to a number of targets. Whenever a hazard is analyzed (whether it was already known as one or whether it was previously a "possible critical state"), the algorithm updates its targets by making them inherit its own sources.

6.4.1 An Example: Inheritance and Loops

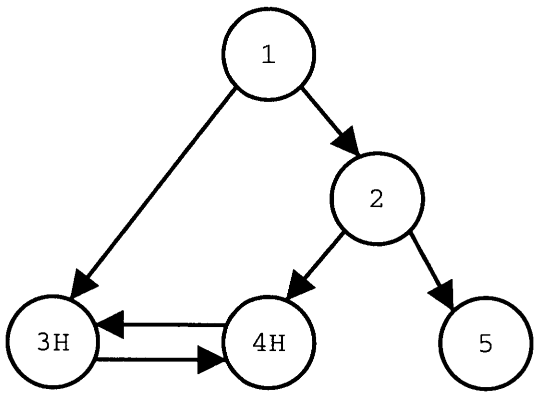

Figure 6-5: Loops and Inheritance Problem

The inheritance algorithm can function even when loops are involved. In Figure 6-5, state 3 has two sources: state 1 and state 4. The list of target $\leftarrow$ source associations is

$3 \leftarrow 1, 4$

State 4 also has two predecessors: state 2 and state 3. It adds state 2 as a source and it inherits state 3's sources. State 1 is thus added as a source, but since state 4 cannot be its own source, that is not added. The list of associations is now

$3 \leftarrow 1, 4$
$4 \leftarrow 1, 2$

The list can also be viewed in source $\rightarrow$ target form as

$1 \rightarrow 3, 4$
$2 \rightarrow 4$
$4 \rightarrow 3$

Finally, state 4 propagates its own sources to its sole target, state 3. First, it deletes itself from state 3's source list, since, as a hazard, it was only a temporary source. Since state 3 already has state 1 as a source, that doesn't need to be added, but state 3 does inherit state 2 as a new source. We thus have the final target $\leftarrow$ source association list:

$3 \leftarrow 1, 2$
$4 \leftarrow 1, 2$

A visual inspection shows that, yes, both states 1 and 2 can lead to state states 3 and 4, and states 1 and 2 could also lead to the safe state 5. States 1 and 2 are thus critical states corresponding to states 3 and 4.

6.5 Dead Ends

There may be hazards in the system with no predecessors. There are two possible reasons for this:

1. The hazard may simply be unreachable from any start state. The algorithm removes it as a source from all of its targets' source lists. Any targets which no longer have any sources are themselves unreachable.
2. The hazard is a hazardous start state. The algorithm notes it as such. Any states which have this hazard as a source keep that association so the user can see how much of the system is affected by this hazardous start state in case the system has multiple start states.

6.6 Running Time

Assume that there are n valid total states. Calculating all cells of the state transition matrix *a priori* would require $O(n^2)$ time. During the execution of the backwards searching hazard elimination analysis algorithm, however, fewer cells need to be calculated. For each hazard and critical state, the algorithm compares the state with every other state to look for predecessors or successors.

Now assume that there are h hazardous states and c critical states. Calculating cells lazily requires only $O(n \cdot (h + c))$ time. This lazy evaluation system can provide significant time savings when $n >> h + c$, which should generally be the case in a large model.

Chapter 7

The Soda Machine Example

The latest release of SpecTRM ships with an Altitude Switch Model (ASW) that is almost entirely input-driven. Hazard analysis does not provide very interesting results on this system, since, for nearly any given state, nearly every other state is a predecessor. Thus, for any given set of hazards, the list of critical states would be just about every other state. More complex models exist which are more input-driven, but their complexity obfuscates the results of hazard analysis. The Soda Machine Example is a heavily state-driven system that serves as an effective testing ground for the hazard analysis. Appendix A details the relevant portions of the SpecTRM model.

7.1 Description of Function

This model represents a soft drink vending machine. The machine has a coin slot which accepts nickels or dimes, and it sports a "dispense" button. Sodas cost (only) ten cents. If the operator inserts any extra money, the machine returns any change immediately, leaving its internal count at 10. The operator must press the dispense button after inserting sufficient funds in order to get a drink. At that point, the machine dispenses a drink and resets the total to zero.

The coin slot weighs the coin and sends a message to the controller every second, stating its current status. The detection mechanism only detects whether a coin is currently in the slot; that is, if a coin remains in the slot for more than a second before falling through, the coin slot may report the same coin twice. In order to allow the software to distinguish this case from two consecutively inserted coins of the same type, a hardware interlock prevents a new coin from falling into the slot for 2 seconds after the previous coin leaves. This means that the slot will send at least one "no coin" to the controller between coins.

Finally, if the software receives invalid inputs from the hardware, the controller turns on an "error" light and ceases to dispense any money or sodas.

7.1.1 Internal Function

Internally, the "Coin Status" state element mirrors the status of the coin slot. When its state transitions from "None" to "Nickel" or "Dime", the "Coin Inserted" state element transitions as well; it remains in this state only until the next transition, however, before returning to the "None" state. The "Total" state element transitions appropriately based on the the last value of "Total", the current value of "Coin Inserted", and incoming messages from the "Dispense" button. Likewise, outputs commands trigger the dispensing of change and sodas based on the internal state.

7.2 Tests

The following discussion will refer to total state indices, which are indices in the list of all valid total states. Appendix B.2 provides the list of valid total states in index order for the Soda Machine model.

7.2.1 Origins of InternalFaultDetected

When the system receives no inputs for over 3 seconds at startup, it enters "Internal-FaultDetected" mode, and the error light is turned on. The system cannot exit this mode; the operator must reboot the system in this case.

The internal state of the system can still change if a foolhardy operator continues to insert money, however. States 27-51 are all in InternalFaultDetected mode, but we need not be concerned with exactly which of them the system is in.

Assume that we want to find out how the system might enter the InternalFaultDetected mode. We can provide it with states 27-51 all marked as hazards. The hazard analysis algorithm provides the following output:

```
Results:

Hazard 27  <- critical states: 0 1
Hazard 28  <- critical states: 0 1
Hazard 29  <- critical states: 0 1
Hazard 30  <- critical states: 0 1
Hazard 31  <- critical states: 0 1
Hazard 32  <- critical states: 0 1
Hazard 33  <- critical states: 0 1
Hazard 34  <- critical states: 0 1
Hazard 35  <- critical states: 0 1
Hazard 36  <- critical states: 0 1
Hazard 37  <- critical states: 0 1
Hazard 38  <- critical states: 0 1
Hazard 39  <- critical states: 0 1
Hazard 40  <- critical states: 0 1
Hazard 41  <- critical states: 0 1
Hazard 42  <- critical states: 0 1
Hazard 43  <- critical states: 0 1
Hazard 44  <- critical states: 0 1
Hazard 45 IS UNREACHABLE!
Hazard 46  <- critical states: 0 1
Hazard 47  <- critical states: 0 1
Hazard 48  <- critical states: 0 1
Hazard 49  <- critical states: 0 1
Hazard 50  <- critical states: 0 1
Hazard 51  <- critical states: 0 1
```

All InternalFaultDetected states can only come from the low-risk states 0 or 1. State 0 is the system start state, and state 1 is the state in which no inputs have been received since system start.

Running a one-step successor search on states 0 and 1 (or by looking down the first two columns of Table B.1), we can see that the successors of both states are states 1, 5, 10, 19, and 27.

We have thus verified that the system is correctly designed in this case, states 0 and 1 may transition to state 27 if 3 seconds have passed without any inputs.

7.2.2 Hazard Discovery

Now consider if we wish to consider any operational state with an element in "Unknown" or "FaultDetected" to be a hazard. We provide the hazard analysis algorithm with the hazardous states {2, 3, 4, 8, 9, 13, 18, 22, 26}:

```
Input Hazards, Valid: (2 3 4 8 9 13 18 22 26) Newly Discovered
Hazards: (17 25)

Results:

Hazard 2  <- critical states: 5 6 7 10 11 12 14 15 16 19 20 21 23 24
Hazard 3  <- critical states: 10 11 12 14 15 16 19 20 21 23 24
Hazard 4  <- critical states: 10 11 12 14 15 16 19 20 21 23 24
Hazard 8  <- critical states: 10 11 12 14 15 16 19 20 21 23 24
Hazard 9  <- critical states: 10 11 12 14 15 16 19 20 21 23 24
Hazard 13  <- critical states: 10 11 12 14 15 16 19 20 21 23 24
Hazard 17  <- critical states: 10 11 12 14 15 16 19 20 21 23 24
Hazard 18  <- critical states: 10 11 12 14 15 16 19 20 21 23 24
Hazard 22  <- critical states: 10 11 12 14 15 16 19 20 21 23 24
Hazard 25  <- critical states: 10 11 12 14 15 16 19 20 21 23 24
Hazard 26  <- critical states: 10 11 12 14 15 16 19 20 21 23 24
```

The algorithm discovers that states 17 and 25 can only transition to other hazardous states, so they must be hazardous themselves. (Unlike most states, they cannot self-transition. In both cases, CoinStatus is not currently "None", so CoinInserted must become something other than "Nickel" or "Dime" after the next transition.)

Chapter 8

Conclusions

This implementation of a backwards searching hazard elimination analysis algorithm is functional and effective. It is capable of directly reading SpecTRM data files, allowing system engineers to use a single model for design and analysis. It can determine the validity of a system state, and it can provide the successors and predecessors of a system state. These features can form the basis of a variety of future algorithms, and this tool is sufficiently modular to make that task easy. Finally, this tool can successfully determine the critical states that correspond to a given set of hazardous states. Along the way, it can discover any hazardous states the user did not provide, and it can handle a variety of challenging scenarios.

The algorithm needs to support additional SpecTRM features before it can analyze larger existing real-world models, but its design should ensure that it would be faster than a forward-searching algorithm that needs to calculate a larger portion of the reachability matrix.

8.1 Future Work

There are two categories of future work necessary for improvement of this tool: expanded support of SpecTRM features and improved interface for the backwards searcher.

Some SpecTRM features used by many real-world models but which were not essential for a demonstration of the algorithm have yet to be implemented. Two SpecTRM features discussed in this paper, macros and input consistency checking, are not yet supported by the tool. Support for some complex features such as timing would also broaden the scope of models that the tool supports.

Currently, the tool only supports the direct input and output of individual, unique total system states. Wild card support, both during input and during output, would improve ease of use and readability when using larger models.

Appendix A

Soda Machine Model

Display Output

FaultDisplay

Destination: ErrorLight

Message: FaultMessage

Acceptable Values: BadCoinSignal

TRIGGERING CONDITION

Operational	T	*
Total in state FaultDetected	T	*
InternalFaultDetected	*	T

MESSAGE CONTENTS

Field	Value
FAULT	BadCoinSignal

Output Command

SodaDispense

Destination: Dispenser

Message: SodaMessage

Acceptable Values: DispenseSoda

TRIGGERING CONDITION

Operational	T
Last Value of Total in state Ten	T
DispenseButton is Pressed	T

MESSAGE CONTENTS

Field	Value
SODA	DispenseSoda

Output Command

ChangeDime

Destination: Dispenser

Message: ChangeMessage

Acceptable Values: Dime

TRIGGERING CONDITION

Operational	T	T
Last Value of Total in state Ten	T	*
CoinInserted in state Dime	T	*
Last value of CoinStatus in state Unknown	*	T
CoinStatus in state Dime	*	T

MESSAGE CONTENTS

Field	Value
CHANGE	Dime

Output Command

ChangeNickel

Destination: Dispenser

Message: ChangeMessage

Acceptable Values: Nickel

TRIGGERING CONDITION

Operational	T	T	T
Last Value of Total in state Five	T	*	*
CoinInserted in state Dime	T	*	*
Last Value of Total in state Ten	*	T	*
CoinInserted in state Nickel	*	T	*
Last value of CoinStatus in state Unknown	*	*	T
CoinStatus in state Nickel	*	*	T

MESSAGE CONTENTS

Field	Value
CHANGE	Nickel

Control Mode

Soda Machine

DEFINITION

= Startup

Power Up	T	*
Startup	*	T
Total in state Unknown	*	T
Time Since Soda Machine Entered Startup > 3 seconds	*	F

= Operational

Startup	T	*
Operational	*	T
Total in state Unknown	F	*

= InternalFaultDetected

InternalFaultDetected	T	*
Startup	*	T
Time Since Soda Machine Entered Startup > 3 seconds	*	T
Total in state Unknown	*	T

Supervisory Mode

Soda Machine Supervisor

DEFINITION

= Soda Controls

Power Up	T	*
Soda Controls	*	T

State Value

CoinStatus

DEFINITION

= Startup

Power Up	T	*
CoinStatusSignal is Obsolete	*	T

= None

CoinStatusSignal is None	T

= Nickel

CoinStatusSignal is Nickel	T

= Dime

CoinStatusSignal is Dime	T

State Value

CoinInserted

DEFINITION

= Unknown

Power Up	T	*
CoinStatus in state Unknown	*	T

= None

Last Value of CoinStatus in state Unknown	T	*	*	*
CoinStatus in state Unknown	F	*	*	*
CoinStatus in state None	*	T	*	*
Last Value of CoinStatus in state Nickel	*	*	T	*
CoinStatus in state Nickel	*	*	T	*
Last Value of CoinStatus in state Dime	*	*	*	T
CoinStatus in state Dime	*	*	*	T
Last Value of CoinInserted in state FaultDetected	F	F	F	F

= Nickel

Last Value of CoinStatus in state None	T
CoinStatus in state Nickel	T
Last Value of CoinInserted in state FaultDetected	F

= Dime

Last Value of CoinStatus in state None	T
CoinStatus in state Dime	T
Last Value of CoinInserted in state FaultDetected	F

= FaultDetected

Last Value of CoinInserted in state FaultDetected	T	*	*
Last Value of CoinStatus in state Dime	*	T	*
CoinStatus in state Nickel	*	T	*
Last Value of CoinStatus in state Nickel	*	*	T
CoinStatus in state Dime	*	*	T

State Value

Total

DEFINITION

= Unknown

Power Up	T	*
CoinInserted in state Unknown	*	T

= Zero

Last Value of Total in state Unknown	T	*	*
Last Value of Total in state Zero	*	T	*
CoinInserted in state None	T	T	*
Last Value of Total in state Ten	*	*	T
DispenseButton is Pressed	*	*	T
CoinInserted in state FaultDetected	*	*	F
CoinInserted in state Unknown	*	*	F
Last Value of Total in state FaultDetected	F	F	F

= Five

Last Value of Total in state Five	T	*
CoinInserted in state None	T	*
Last Value of Total in state Zero	*	T
CoinInserted in state Nickel	*	T
Last Value of Total in state FaultDetected	F	F

= Ten

Last Value of Total in state Ten	T	*	*	*
Last Value of Total in state Five	*	T	*	T
CoinInserted in state Nickel	*	T	*	*
Last Value of Total in state Zero	*	*	T	*
CoinInserted in state Dime	*	*	T	T
DispenseButton is Pressed	F	*	*	*
CoinInserted in state FaulDetected	F	*	*	*
CoinInserted in state Unknown	F	*	*	*
Last Value of Total in state FaultDetected	F	F	F	F

= FaultDetected

Last Value of Total in state FaultDetected	T	*	*	*
CoinInserted in state FaulDetected	*	T	*	*
Last Value of Total in state Ten	*	*	T	T
CoinInserted in state Nickel	*	*	T	*
CoinInserted in state Dime	*	*	*	T

Other Elements

Inputs:

CoinStatusSignal {None, Nickel, Dime}

DispenseButton {Released, Pressed}

Messages:

DispenseButtonMessage

SodaMessage

ChangeMessage

FaultMessage

Devices:

CoinSlot

Dispenser

ErrorLight

Appendix B

Soda Machine State Transition Matrix

B.1 State Transition Matrix

Figure B.1 is the state transition matrix for the Soda Machine example. Rows are destinations and columns are sources. Section B.2 contains the list of valid total states that form the axes of this matrix.

```
Key: "1" = reachable, "." = unreachable

 0         1         2         3         4         5
0....................................................
 11..................................................
 ..1111111111111111111111111.........................
 ...11...11...1...11...1..11.........................
 ....1....1........1.......1.........................
 111..1.1..1.1.1.1..1.1.11...........................
 ......1....1...1....1...............................
 .......1....1...1....1..1...........................
 ...1....1....1...1....1..1..........................
 ....1....1........1.......1.........................
1111.......1.1.1.1...................................
 ...........1...1....................................
 ............1...1...................................
 ...1.........1...1..................................
 .......1............................................
 .....1..............................................
 ......11............................................
 ........1...........................................
 ....1....1........111111111.........................
 111................1.1.11...........................
2....................1...............................
 .....................1..1...........................
 ...1..................1..1..........................
 .......1............................................
 .....111............................................
 ........1...........................................
 ....1....1111111111.......1.........................
 11.........................1111111111111111111111111
 ............................11...11...1...11...1..11
 .............................1....1........1.......1
3...........................1..1.1..1.1.1.1..1.1.11..
 ...............................1....1...1....1......
 ................................1....1...1....1..1..
 ............................1....1....1...1....1..1.
 .............................1....1........1.......1
 ...........................1.......1.1.1.1..........
 ....................................1...1...........
 .....................................1...1..........
 ............................1.........1...1.........
 ................................1...................
4..............................1.....................
 ...............................11...................
 .................................1..................
 .............................1....1........111111111
 ...........................1................1.1.11..
 .............................................1......
 ..............................................1..1..
 ............................1..................1..1.
 ................................1...................
 ..............................111...................
5.................................1..................
 .............................1....1111111111.......1
```

Table B.1: Soda Machine State Transition Matrix

B.2 Total State List

```
***** Total State 0 *****

  Power Up True
  Mode  (0,0) MODE #0:Soda Machine in #0:Startup
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,0) STATE #0:CoinStatus in #0:Unknown
  State (1,0) STATE #1:CoinInserted in #0:Unknown
  State (2,0) STATE #2:Total in #0:Unknown

***** Total State 1 *****

  Power Up False
  Mode  (0,0) MODE #0:Soda Machine in #0:Startup
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,0) STATE #0:CoinStatus in #0:Unknown
  State (1,0) STATE #1:CoinInserted in #0:Unknown
  State (2,0) STATE #2:Total in #0:Unknown

***** Total State 2 *****

  Power Up False
  Mode  (0,1) MODE #0:Soda Machine in #1:Operational
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,0) STATE #0:CoinStatus in #0:Unknown
  State (1,0) STATE #1:CoinInserted in #0:Unknown
  State (2,0) STATE #2:Total in #0:Unknown

***** Total State 3 *****

  Power Up False
  Mode  (0,1) MODE #0:Soda Machine in #1:Operational
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,0) STATE #0:CoinStatus in #0:Unknown
  State (1,0) STATE #1:CoinInserted in #0:Unknown
  State (2,4) STATE #2:Total in #4:FaultDetected

***** Total State 4 *****

  Power Up False
  Mode  (0,1) MODE #0:Soda Machine in #1:Operational
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,0) STATE #0:CoinStatus in #0:Unknown
  State (1,4) STATE #1:CoinInserted in #4:FaultDetected
  State (2,4) STATE #2:Total in #4:FaultDetected
```

```
***** Total State 5 *****

  Power Up False
  Mode  (0,1) MODE #0:Soda Machine in #1:Operational
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,1) STATE #0:CoinStatus in #1:None
  State (1,1) STATE #1:CoinInserted in #1:None
  State (2,1) STATE #2:Total in #1:Zero

***** Total State 6 *****

  Power Up False
  Mode  (0,1) MODE #0:Soda Machine in #1:Operational
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,1) STATE #0:CoinStatus in #1:None
  State (1,1) STATE #1:CoinInserted in #1:None
  State (2,2) STATE #2:Total in #2:Five

***** Total State 7 *****

  Power Up False
  Mode  (0,1) MODE #0:Soda Machine in #1:Operational
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,1) STATE #0:CoinStatus in #1:None
  State (1,1) STATE #1:CoinInserted in #1:None
  State (2,3) STATE #2:Total in #3:Ten

***** Total State 8 *****

  Power Up False
  Mode  (0,1) MODE #0:Soda Machine in #1:Operational
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,1) STATE #0:CoinStatus in #1:None
  State (1,1) STATE #1:CoinInserted in #1:None
  State (2,4) STATE #2:Total in #4:FaultDetected

***** Total State 9 *****

  Power Up False
  Mode  (0,1) MODE #0:Soda Machine in #1:Operational
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,1) STATE #0:CoinStatus in #1:None
  State (1,4) STATE #1:CoinInserted in #4:FaultDetected
  State (2,4) STATE #2:Total in #4:FaultDetected

***** Total State 10 *****

  Power Up False
  Mode  (0,1) MODE #0:Soda Machine in #1:Operational
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,2) STATE #0:CoinStatus in #2:Nickel
  State (1,1) STATE #1:CoinInserted in #1:None
  State (2,1) STATE #2:Total in #1:Zero
```

```
***** Total State 11 *****

  Power Up False
  Mode  (0,1) MODE #0:Soda Machine in #1:Operational
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,2) STATE #0:CoinStatus in #2:Nickel
  State (1,1) STATE #1:CoinInserted in #1:None
  State (2,2) STATE #2:Total in #2:Five

***** Total State 12 *****

  Power Up False
  Mode  (0,1) MODE #0:Soda Machine in #1:Operational
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,2) STATE #0:CoinStatus in #2:Nickel
  State (1,1) STATE #1:CoinInserted in #1:None
  State (2,3) STATE #2:Total in #3:Ten

***** Total State 13 *****

  Power Up False
  Mode  (0,1) MODE #0:Soda Machine in #1:Operational
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,2) STATE #0:CoinStatus in #2:Nickel
  State (1,1) STATE #1:CoinInserted in #1:None
  State (2,4) STATE #2:Total in #4:FaultDetected

***** Total State 14 *****

  Power Up False
  Mode  (0,1) MODE #0:Soda Machine in #1:Operational
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,2) STATE #0:CoinStatus in #2:Nickel
  State (1,2) STATE #1:CoinInserted in #2:Nickel
  State (2,1) STATE #2:Total in #1:Zero

***** Total State 15 *****

  Power Up False
  Mode  (0,1) MODE #0:Soda Machine in #1:Operational
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,2) STATE #0:CoinStatus in #2:Nickel
  State (1,2) STATE #1:CoinInserted in #2:Nickel
  State (2,2) STATE #2:Total in #2:Five

***** Total State 16 *****

  Power Up False
  Mode  (0,1) MODE #0:Soda Machine in #1:Operational
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,2) STATE #0:CoinStatus in #2:Nickel
  State (1,2) STATE #1:CoinInserted in #2:Nickel
  State (2,3) STATE #2:Total in #3:Ten
```

```
***** Total State 17 *****

  Power Up False
  Mode  (0,1) MODE #0:Soda Machine in #1:Operational
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,2) STATE #0:CoinStatus in #2:Nickel
  State (1,2) STATE #1:CoinInserted in #2:Nickel
  State (2,4) STATE #2:Total in #4:FaultDetected

***** Total State 18 *****

  Power Up False
  Mode  (0,1) MODE #0:Soda Machine in #1:Operational
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,2) STATE #0:CoinStatus in #2:Nickel
  State (1,4) STATE #1:CoinInserted in #4:FaultDetected
  State (2,4) STATE #2:Total in #4:FaultDetected

***** Total State 19 *****

  Power Up False
  Mode  (0,1) MODE #0:Soda Machine in #1:Operational
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,3) STATE #0:CoinStatus in #3:Dime
  State (1,1) STATE #1:CoinInserted in #1:None
  State (2,1) STATE #2:Total in #1:Zero

***** Total State 20 *****

  Power Up False
  Mode  (0,1) MODE #0:Soda Machine in #1:Operational
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,3) STATE #0:CoinStatus in #3:Dime
  State (1,1) STATE #1:CoinInserted in #1:None
  State (2,2) STATE #2:Total in #2:Five

***** Total State 21 *****

  Power Up False
  Mode  (0,1) MODE #0:Soda Machine in #1:Operational
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,3) STATE #0:CoinStatus in #3:Dime
  State (1,1) STATE #1:CoinInserted in #1:None
  State (2,3) STATE #2:Total in #3:Ten

***** Total State 22 *****

  Power Up False
  Mode  (0,1) MODE #0:Soda Machine in #1:Operational
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,3) STATE #0:CoinStatus in #3:Dime
  State (1,1) STATE #1:CoinInserted in #1:None
  State (2,4) STATE #2:Total in #4:FaultDetected
```

```
***** Total State 23 *****

  Power Up False
  Mode  (0,1) MODE #0:Soda Machine in #1:Operational
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,3) STATE #0:CoinStatus in #3:Dime
  State (1,3) STATE #1:CoinInserted in #3:Dime
  State (2,1) STATE #2:Total in #1:Zero

***** Total State 24 *****

  Power Up False
  Mode  (0,1) MODE #0:Soda Machine in #1:Operational
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,3) STATE #0:CoinStatus in #3:Dime
  State (1,3) STATE #1:CoinInserted in #3:Dime
  State (2,3) STATE #2:Total in #3:Ten

***** Total State 25 *****

  Power Up False
  Mode  (0,1) MODE #0:Soda Machine in #1:Operational
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,3) STATE #0:CoinStatus in #3:Dime
  State (1,3) STATE #1:CoinInserted in #3:Dime
  State (2,4) STATE #2:Total in #4:FaultDetected

***** Total State 26 *****

  Power Up False
  Mode  (0,1) MODE #0:Soda Machine in #1:Operational
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,3) STATE #0:CoinStatus in #3:Dime
  State (1,4) STATE #1:CoinInserted in #4:FaultDetected
  State (2,4) STATE #2:Total in #4:FaultDetected

***** Total State 27 *****

  Power Up False
  Mode  (0,2) MODE #0:Soda Machine in #2:InternalFaultDetected
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,0) STATE #0:CoinStatus in #0:Unknown
  State (1,0) STATE #1:CoinInserted in #0:Unknown
  State (2,0) STATE #2:Total in #0:Unknown

***** Total State 28 *****

  Power Up False
  Mode  (0,2) MODE #0:Soda Machine in #2:InternalFaultDetected
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,0) STATE #0:CoinStatus in #0:Unknown
  State (1,0) STATE #1:CoinInserted in #0:Unknown
  State (2,4) STATE #2:Total in #4:FaultDetected
```

```
***** Total State 29 *****

  Power Up False
  Mode  (0,2) MODE #0:Soda Machine in #2:InternalFaultDetected
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,0) STATE #0:CoinStatus in #0:Unknown
  State (1,4) STATE #1:CoinInserted in #4:FaultDetected
  State (2,4) STATE #2:Total in #4:FaultDetected

***** Total State 30 *****

  Power Up False
  Mode  (0,2) MODE #0:Soda Machine in #2:InternalFaultDetected
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,1) STATE #0:CoinStatus in #1:None
  State (1,1) STATE #1:CoinInserted in #1:None
  State (2,1) STATE #2:Total in #1:Zero

***** Total State 31 *****

  Power Up False
  Mode  (0,2) MODE #0:Soda Machine in #2:InternalFaultDetected
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,1) STATE #0:CoinStatus in #1:None
  State (1,1) STATE #1:CoinInserted in #1:None
  State (2,2) STATE #2:Total in #2:Five

***** Total State 32 *****

  Power Up False
  Mode  (0,2) MODE #0:Soda Machine in #2:InternalFaultDetected
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,1) STATE #0:CoinStatus in #1:None
  State (1,1) STATE #1:CoinInserted in #1:None
  State (2,3) STATE #2:Total in #3:Ten

***** Total State 33 *****

  Power Up False
  Mode  (0,2) MODE #0:Soda Machine in #2:InternalFaultDetected
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,1) STATE #0:CoinStatus in #1:None
  State (1,1) STATE #1:CoinInserted in #1:None
  State (2,4) STATE #2:Total in #4:FaultDetected

***** Total State 34 *****

  Power Up False
  Mode  (0,2) MODE #0:Soda Machine in #2:InternalFaultDetected
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,1) STATE #0:CoinStatus in #1:None
  State (1,4) STATE #1:CoinInserted in #4:FaultDetected
  State (2,4) STATE #2:Total in #4:FaultDetected
```

```
***** Total State 35 *****

  Power Up False
  Mode  (0,2) MODE #0:Soda Machine in #2:InternalFaultDetected
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,2) STATE #0:CoinStatus in #2:Nickel
  State (1,1) STATE #1:CoinInserted in #1:None
  State (2,1) STATE #2:Total in #1:Zero

***** Total State 36 *****

  Power Up False
  Mode  (0,2) MODE #0:Soda Machine in #2:InternalFaultDetected
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,2) STATE #0:CoinStatus in #2:Nickel
  State (1,1) STATE #1:CoinInserted in #1:None
  State (2,2) STATE #2:Total in #2:Five

***** Total State 37 *****

  Power Up False
  Mode  (0,2) MODE #0:Soda Machine in #2:InternalFaultDetected
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,2) STATE #0:CoinStatus in #2:Nickel
  State (1,1) STATE #1:CoinInserted in #1:None
  State (2,3) STATE #2:Total in #3:Ten

***** Total State 38 *****

  Power Up False
  Mode  (0,2) MODE #0:Soda Machine in #2:InternalFaultDetected
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,2) STATE #0:CoinStatus in #2:Nickel
  State (1,1) STATE #1:CoinInserted in #1:None
  State (2,4) STATE #2:Total in #4:FaultDetected

***** Total State 39 *****

  Power Up False
  Mode  (0,2) MODE #0:Soda Machine in #2:InternalFaultDetected
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,2) STATE #0:CoinStatus in #2:Nickel
  State (1,2) STATE #1:CoinInserted in #2:Nickel
  State (2,1) STATE #2:Total in #1:Zero

***** Total State 40 *****

  Power Up False
  Mode  (0,2) MODE #0:Soda Machine in #2:InternalFaultDetected
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,2) STATE #0:CoinStatus in #2:Nickel
  State (1,2) STATE #1:CoinInserted in #2:Nickel
  State (2,2) STATE #2:Total in #2:Five
```

```
***** Total State 41 *****

  Power Up False
  Mode  (0,2) MODE #0:Soda Machine in #2:InternalFaultDetected
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,2) STATE #0:CoinStatus in #2:Nickel
  State (1,2) STATE #1:CoinInserted in #2:Nickel
  State (2,3) STATE #2:Total in #3:Ten

***** Total State 42 *****

  Power Up False
  Mode  (0,2) MODE #0:Soda Machine in #2:InternalFaultDetected
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,2) STATE #0:CoinStatus in #2:Nickel
  State (1,2) STATE #1:CoinInserted in #2:Nickel
  State (2,4) STATE #2:Total in #4:FaultDetected

***** Total State 43 *****

  Power Up False
  Mode  (0,2) MODE #0:Soda Machine in #2:InternalFaultDetected
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,2) STATE #0:CoinStatus in #2:Nickel
  State (1,4) STATE #1:CoinInserted in #4:FaultDetected
  State (2,4) STATE #2:Total in #4:FaultDetected

***** Total State 44 *****

  Power Up False
  Mode  (0,2) MODE #0:Soda Machine in #2:InternalFaultDetected
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,3) STATE #0:CoinStatus in #3:Dime
  State (1,1) STATE #1:CoinInserted in #1:None
  State (2,1) STATE #2:Total in #1:Zero

***** Total State 45 *****

  Power Up False
  Mode  (0,2) MODE #0:Soda Machine in #2:InternalFaultDetected
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,3) STATE #0:CoinStatus in #3:Dime
  State (1,1) STATE #1:CoinInserted in #1:None
  State (2,2) STATE #2:Total in #2:Five

***** Total State 46 *****

  Power Up False
  Mode  (0,2) MODE #0:Soda Machine in #2:InternalFaultDetected
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,3) STATE #0:CoinStatus in #3:Dime
  State (1,1) STATE #1:CoinInserted in #1:None
  State (2,3) STATE #2:Total in #3:Ten
```

```
***** Total State 47 *****

  Power Up False
  Mode  (0,2) MODE #0:Soda Machine in #2:InternalFaultDetected
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,3) STATE #0:CoinStatus in #3:Dime
  State (1,1) STATE #1:CoinInserted in #1:None
  State (2,4) STATE #2:Total in #4:FaultDetected

***** Total State 48 *****

  Power Up False
  Mode  (0,2) MODE #0:Soda Machine in #2:InternalFaultDetected
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,3) STATE #0:CoinStatus in #3:Dime
  State (1,3) STATE #1:CoinInserted in #3:Dime
  State (2,1) STATE #2:Total in #1:Zero

***** Total State 49 *****

  Power Up False
  Mode  (0,2) MODE #0:Soda Machine in #2:InternalFaultDetected
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,3) STATE #0:CoinStatus in #3:Dime
  State (1,3) STATE #1:CoinInserted in #3:Dime
  State (2,3) STATE #2:Total in #3:Ten

***** Total State 50 *****

  Power Up False
  Mode  (0,2) MODE #0:Soda Machine in #2:InternalFaultDetected
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,3) STATE #0:CoinStatus in #3:Dime
  State (1,3) STATE #1:CoinInserted in #3:Dime
  State (2,4) STATE #2:Total in #4:FaultDetected

***** Total State 51 *****

  Power Up False
  Mode  (0,2) MODE #0:Soda Machine in #2:InternalFaultDetected
  Mode  (1,0) MODE #1:Soda Machine Supervisor in #0:Soda Controls
  State (0,3) STATE #0:CoinStatus in #3:Dime
  State (1,4) STATE #1:CoinInserted in #4:FaultDetected
  State (2,4) STATE #2:Total in #4:FaultDetected
```

Bibliography

[1] Étienne Gagnon. SableCC, An Object-Oriented Compiler Framework. Master's thesis, McGill University, Montreal, Department of Computer Science, March 1998.

[2] Nancy Leveson. Intent specifications: An approach to building human-centered specifications. In *IEEE Transactions on Software Engineering*, January 2000.

[3] Natasha Anita Neogi. *Hazard Elimination Using Backwards Reachability Techniques in Discrete and Hybrid Models*. PhD thesis, MIT, Department of Aeronautics and Astronautics, December 1988.

[4] Safeware Engineering Corporation. *SpecTRM User Manual*, 2001.

[5] Safeware Engineering Corporation. *SpecTRM XML Schema Documentation*, 2001.