INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

AUTOMATING TEST GENERATION FOR DISCRETE EVENT ORIENTED REAL-TIME EMBEDDED SYSTEMS

by

Steven J Cunning

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2000

UMI Number: 9992141

Bell & Howell Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

THE UNIVERSITY OF ARIZONA ®
GRADUATE COLLEGE

As members of the Final Examination Committee, we certify that we have read the dissertation prepared by Steven J Cunning

entitled Automating Test Generation for Discrete Event Oriented Real-Time Embedded Systems

and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy

Jerzy W. Rozenblit	October 19, 2000 **Date**
Bernard P. Zeigler	October 19, 2000 **Date**
JoDale Carothers	October 19, 2000 **Date**
William E. Bracker	October 19, 2000 **Date**
	Date

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copy of the dissertation to the Graduate College.

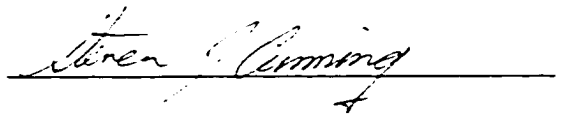
I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Dissertation Director Jerzy W. Rozenblit	October 19, 2000 **Date**

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the copyright holder.

SIGNED: 

ACKNOWLEDGMENTS

I would like to thank my wife Kathleen for the support she has given me. I would also like to thank my children, Savannah and Daylon, for their patience. My gratitude also goes out to my parents, Charles and Barbara, for their continued encouragement.

I would like to thank Dr. Jerzy Rozenblit for providing me with the opportunity to pursue this degree and for the open and relaxed environment in which this work was completed. I appreciate the time and efforts of my committee members; Dr. Bernard Zeigler, Dr. Jo Dale Carothers, Dr. Greg Andrews, Dr. William Bracker, and Dr. Fredrick Hill.

I would not have been able to pursue this degree without the support of my management at Raytheon (Hughes at the time I started). Special thanks goes to Lynne Bracker who has supported and encouraged me from start to finish.

This work could not have been completed without the support and input from the members of our Model-based Design Lab; Stephan Schulz, Tony Ewing, and John Olson. In addition, Yarisa Jaroch and Pallav Gupta have given considerable contributions by implementing several portions of the test generation process described in this work. My thanks also goes to Dr. Hessam Sarjoughian for his insights and helpful suggestions.

I sincerely appreciate the members of the Naval Research Labs, Constance Heitmeyer, Bruce Labaw, Jim Kirby, Ralph Jeffords, and Todd Grimm, for providing me their SCR Toolset and for their very quick responses when questions arose.

This work was supported by grant number 9554561 from the National Science Foundation and by a Raytheon/Hughes Fellowship.

TABLE OF CONTENTS

TABLE OF CONTENTS - *Continued*

TABLE OF CONTENTS - *Continued*

TABLE OF CONTENTS - *Continued*

LIST OF FIGURES

LIST OF FIGURES - *Continued*

LIST OF TABLES

ABSTRACT

The purpose of this work is to provide a method for the automatic generation of test scenarios from the behavioral requirements of a system. The goal of the generated suite of test scenarios is to allow the system design to be validated against the requirements. The benefits of automatic test generation include improved efficiency, completeness (coverage), and objectivity (removal of human bias).

The Model-based Codesign method is refined by defining a design process flow. This process flow includes the generation of test suites from requirements and the application of these tests across multiple levels of the design path.

An approach is proposed that utilizes what is called a requirements model and a set of four algorithms. The requirements model is an executable model of the proposed system defined in a deterministic state-based modeling formalism. Each action in the requirements model that changes the state of the model is identified with a unique requirement identifier. The scenario generation algorithms perform controlled simulations of the requirements model in order to generate a suite of test scenarios applicable for black box testing.

A process defining the generation and use of the test scenarios is developed. This process also includes the treatment of temporal requirements which are considered separately from the generation of the test scenarios. An algorithm is defined to combine the test scenarios with the environmental temporal requirements to produce timed test scenarios in the IEEE standard C/ATLAS test language. An algorithm is also defined to describe the behavior of the test environment as it interprets and applies the C/ATLAS

test programs. Finally, an algorithm to analyze the test results logged while applying the test scenario is defined.

Measurements of several metrics on the scenario generation algorithms have been collected using prototype tools. The results support the position that the algorithms are performing reasonably well, that the generated test scenarios are adequately efficient, and that the processing time needed for test generation grows slowly enough to support much larger systems.

1 INTRODUCTION

As developers of computer-based systems, how can we be sure our systems function and perform as they should? This broad statement summarizes the general issue being addressed in this work.

Let's take a closer look at this question. First, what is a *computer-based system*? For our purposes this is any system that contains at least one digital processing element. This could be a system with a single small controller such as might be found in an electric razor, or may be complex system with multiple processing elements either co-located or distributed over some geographic distance such as a coordinated group of traffic signals. Second, what is meant by *function and perform*? To function correctly, the system must generate the correct outputs for all expected inputs without regard for the time needed to generate the output. To perform correctly, the outputs must be generated within the proper amount of time. In other words, for the system to be correct it must both generate the proper outputs (function) and do so within the proper timing constraints (performance). Generally, the term behavior is used to describe the simultaneous consideration of function and performance.

Finally, what is meant by the vague phrase *as it should*? This is a difficult question to answer. There are two aspects to consider. The first is whether the system behaves as expected and the second is whether the expectations are correct. This is exactly analogous to the common meanings given to Verification and Validation in the domain of modeling [1]. The common questions are, "Did I build the system right?" and "Did I build the right system?" The first question is similar in both system design *and* modeling

and deals with whether the resulting system behaves as intended. This is usually treated by informal testing during system development. The second question must be handled differently for system design. In the modeling domain, the effort is usually directed toward developing a computer model of a real word system. In this case, the second question can be answered by comparing the model to the real system. For system design, the effort is to develop a system that does not yet exist. So what defines the correct system behavior? This role is assumed by the system requirements. The requirement statements for a system should define the behavior of the system so that candidate designs can be measured against them.

In order to advance our ability to answer the question above, the primary contribution of this work is the development of a method for the automatic generation of test scenarios. This method is based on state-based modeling and simulation. Multiple algorithms with graph and computation theoretic considerations have been developed. In addition, process flows have been developed that define how the scenario generation and associated algorithms fit into Model-based Codesign.

1.1 Motivation

Much work in the area of requirements engineering has been done over the past twenty years. Many languages and methods have been developed and implemented [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]. White [13] provides a comparative analysis of eight such methods. Throughout these works, the problem statement, which has remained essentially unchanged over the years, is that incomplete, ambiguous, incompatible, and incomprehensible requirements lead to poor designs. The conclusions have also been

consistent. The use of a structured requirements language, usually with tool support and within a requirements solicitation and documentation process leads to early detection of problems and misunderstandings. The resolution of which leads to better designs.

The question raised by personal experience and echoed in the literature [14, 15] is: "Why aren't these languages and methods in wide spread use today?" Many valid reasons are given, one of which is that management and engineering both need to believe in the necessity of the requirements engineering effort and in the benefits of the end product, the requirements specification. Another factor that may be hindering the acceptance of worthy improvements is that they present too large of a leap for many organizations to take. In other words, improvements are developed without paying enough attention to the state of practice, and then it is hoped that they will catch on. It is our position that a better approach would be to build on current practice, i.e., languages, tools, and methods in use today. This will more easily guide industrial organizations toward improved requirements engineering and the resulting benefits.

While the idea of automatic generation of test scenarios from requirements specifications is not new, acceptance of the proposed methods have not been embraced in industry. Most of these methods rely on the development of a finite state machine (FSM) based representation of the system requirements [9, 10, 18, 40, 41, 43, 45, 99, 100], which we will call the requirements model. These models are evaluated in order to derive sequences of stimuli (applied to the system) and responses (expected from the system).

What we feel is necessary for industry acceptance, is that industrial management must believe that the benefits of the requirements models outweigh the cost to develop them. In order for this to happen, at least one necessary aspect is the availability of tools that allow quick and easy development of these models. These tools must be available and, at least to some extent, presently in use by industry. Examples of two such tools are Statemate [16] and the SCR (Software Cost Reduction) tool set [17].

Using a language and tool set presently in use allows industrial organization to more easily adopt or expand their use. Automatic test generation will be adopted if adequate and flexible tool support that is easily integrated with the requirements modeling tools is available. The automatic test generation must provide effective test scenarios for a wide range of systems and test objectives. Flexibility to allow an organization to control the type of test coverage desired should also be included.

The current practice of test generation in industry is largely ad hoc [18]. One of the benefits of automation is to provide consistent coverage of system requirements and to generate test scenarios in an objective and unbiased manner. Another advantage of automation is relieving the designers or test engineers of much of the tediousness of this task. And finally, automation will reduce the time needed for test generation. This will allow an organization using automatic test generation methods to more easily handle the inevitable changes in requirements.

Our motivation comes from a systems perspective. In particular, Model-based Codesign of real-time embedded systems [19, 20]. This method relies on system models at increasing levels of fidelity in order to explore design alternatives and to evaluate the

correctness of these designs. As a result, the tests that we desire should cover all system requirements in order to determine if all requirements have been implemented in the design. The set of generated tests will then be maintained and applied to system models of increasing fidelity and to the system prototype in order to verify the consistency between models and physical realizations.

1.2 Context

Although it should be possible to apply the algorithms developed in this work to any design process using state-based requirements and design models, they have been developed within a particular context. This context will be described in the following sections.

1.2.1 Real-Time Embedded Systems

This work focuses on embedded computer-based systems with real-time deadlines [21, 22, 23]. These are computer systems without general user interfaces and usually perform a dedicated task. Real-time deadlines are temporal constraint requirements and failure to meet these requirements constitutes a system failure. These systems are generally referred to as hard real-time embedded systems.

1.2.2 Codesign

Codesign is a term that refers to a design process where computer-based systems (particularly embedded systems) are designed within a process that keeps the hardware and software design activities tightly coupled. This research area has sprouted from a

practical need to improve the traditional design process and has gained considerable attention in the past several years.

The design of embedded systems has traditionally been carried out by a) decomposing and allocating the system to hardware and software, b) allowing separate hardware and software design teams to design their respective portions of the system, and then c) integrating the hardware and software. This separation of design tasks leads to the potential for initial design mistakes to be carried through until the integration phase where they are much more difficult and costly to correct.

Numerous authors point to the deficiencies of the traditional hardware/software design frameworks [20, 24, 25, 26, 27, 28]. They strongly advocate a process that fosters the integration of hardware and software design perspectives. Therefore, a unified representation is needed for modeling a system independent of its physical realization.

The approaches to hardware/software codesign are many and varied. The primary goal of research in this area is to improve the design process by reasoning about the design at the systems level (i.e., no allocation or assumptions regarding implementation of functional components in hardware or software) and to more tightly couple the hardware and software design efforts. Design correctness, as measured against design requirements, and design alternatives are evaluated at the systems level. The primary means to mitigate the risk of having separate hardware and software design efforts are; common design models, hardware/software co-simulation, or hardware and software design synthesis from system level models (i.e., rapid prototyping).

1.2.3 Model-based Codesign

The variant of codesign under development at the University of Arizona has been termed *Model-based Codesign* [19, 20, 28]. This design process uses stepwise refinement of simulatable models and offers the opportunity to abstract system components at multiple levels of representation. In this methodology, a set of requirements and constraints is obtained for the system to be modeled. The system is then described as an abstract model that is a combination of its structural and associated behavioral specifications. Model components are specified at a high level of abstraction to remain implementation independent.

In Model-based Codesign, we verify correctness of models through computer simulation. A simulation test setup is called an *experimental frame* [29, 30, 31]. It is associated with the system's model during simulation. Such frames specify conditions under which the model of the system is observed. Simulation is then executed according to the run conditions prescribed by the frames. At the end of the simulation process a virtual system prototype is obtained. The design is then partitioned into hardware, software and corresponding interfaces using a process that we call *model mapping*.

The Model-based Codesign process is illustrated in Figure 1. The process begins with functional and behavioral definition of requirements. The output of this step feeds both the design and test process flows. These two activities may proceed in parallel.

The design flow starts with the development of an executable design model. The first loop in the design flow is a functional simulation and model refinement loop, which

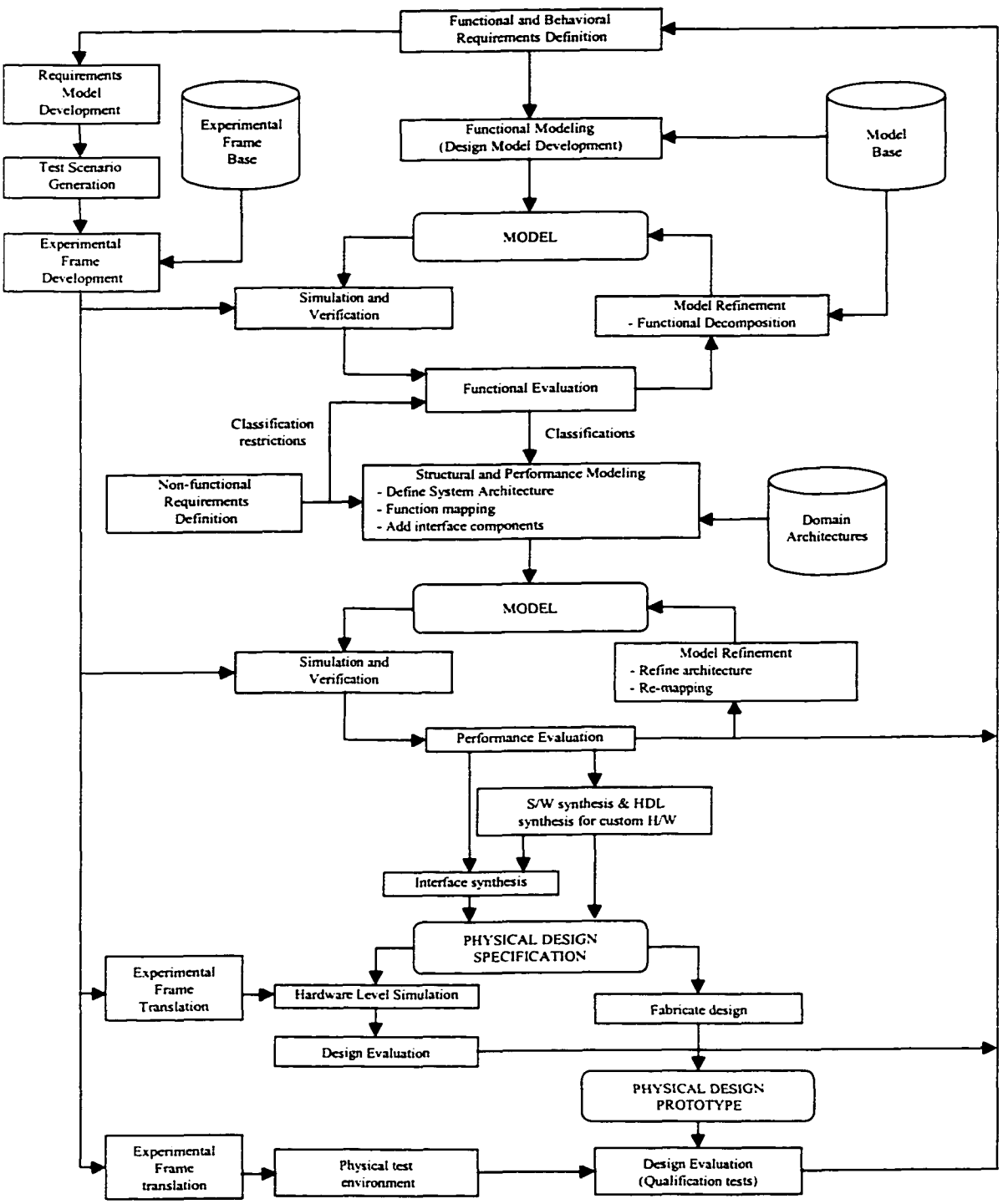


Figure 1. Model-based Codesign Process

is used to iteratively refine the design model until it is functionally correct. When a model has been verified to be functionally correct, the structural definition of requirements and modeling takes place. This step relates physical design constraints to a proposed physical architecture and enhances the model with performance estimates for computation and communication based upon the proposed physical architecture. The second loop in the design flow is a performance simulation and model refinement loop, which is used to refine the system architecture based on model performance. When the simulations of a design meet the system functional (first loop) and performance requirements (second loop), the synthesis and implementation step may begin. This activity involves extracting design information from the models in order to produce a physical prototype.

The test and experimental frame development flow involves the creation of a set of test scenarios based upon the system requirements. These test scenarios are used to create a suite of experimental frames to support testing at the virtual level. The experimental frames are translated to support testing at lower levels of the design process flow.

1.2.4 Testing in Model-based Codesign

Model-based Codesign relies heavily on modeling and simulation to evaluate potential system designs. Efficient use of simulation requires prepared sets of system stimuli and a means for evaluating the system responses to them. Test scenarios must consist of a time ordered sequence of events representing both the stimuli to be applied to

the system and the expected responses (referred to as a test scenario). The proposed test generation process is illustrated in Figure 2.

It is expected that any design project will start with a document stating the system requirements in a textual form. The textual requirements will be modified to include a unique identifier for each requirement if such identifiers do not already exist. The requirements model is then developed from the textual representation. This model represents the system in the form of a finite state machine, in a formal modeling language. The internal structure of this model is not of primary importance, as long as the external behavior is complete and consistent with the original requirements. This model also provides a virtual prototype of the system which can be used to gain insight into the system's dynamic behavior. This can be invaluable in correcting and clarifying the requirements.

In addition to capturing the behavior of the system, the requirements modeling language must support the annotation of interface requirements and requirements identifiers. Interface requirements define the required behavior of the system's environment and temporal requirements for both the system and environment. The requirements identifiers provide traceablility between the model and the textual requirements and are used by the test scenario generator to determine requirements coverage.

After development of the requirements model, the interface constraints and system functions are extracted and synthesized into a high level programming language. This

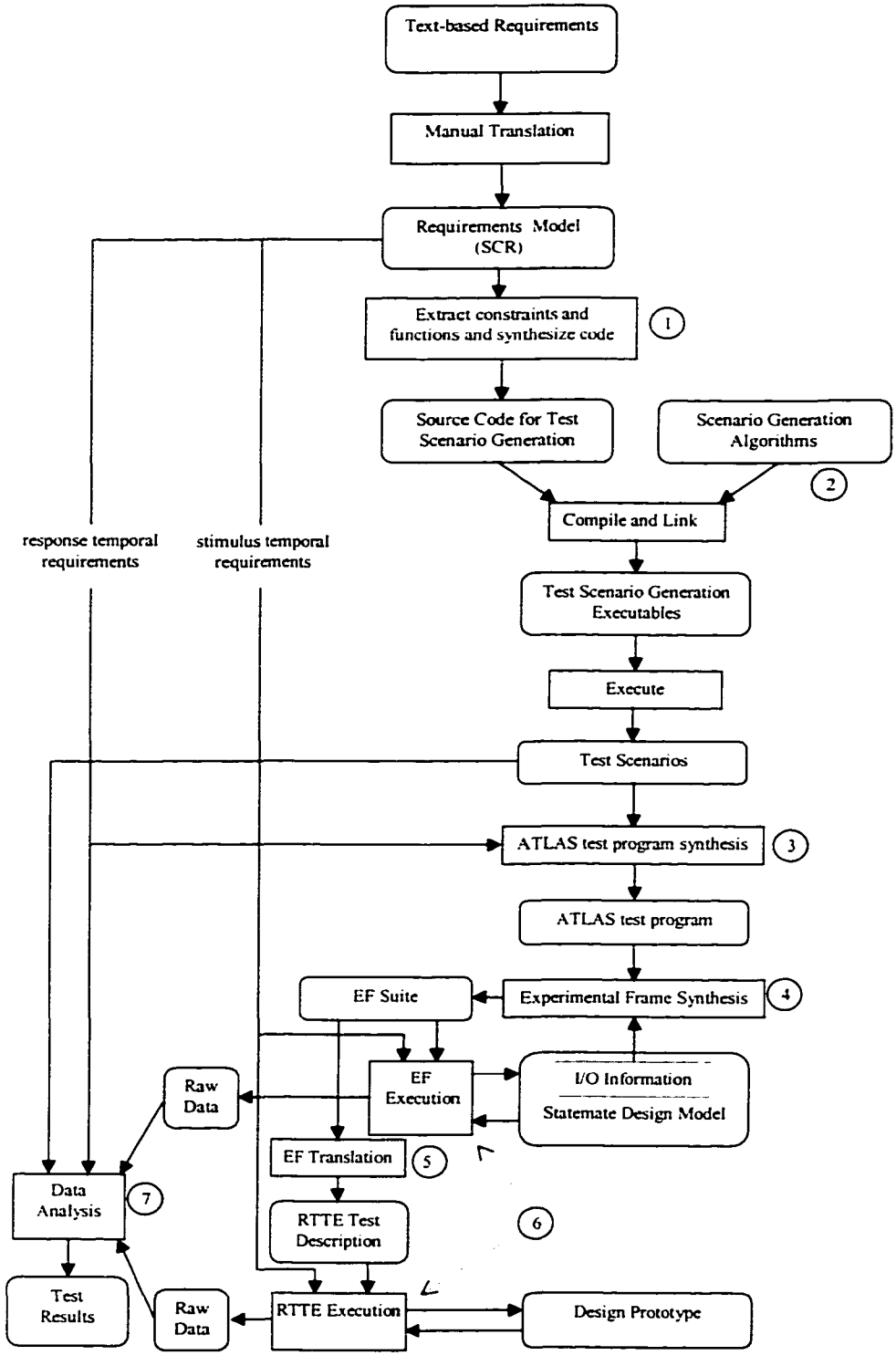


Figure 2. Scenario Generation and Use

code is compiled with the test scenario generation algorithm to produce the test scenario generator. The test scenario generator uses the system functions, interface constraints if any and manually selected test input values to perform a controlled simulation of the requirements model. This simulation performs a state space exploration of the model by starting from the specified initial state and applying available system stimuli in order to generate new system states. The details of scenario generation are given in chapters 2 and 3.

The output produced by executing the test scenario generator is a rooted tree called the scenario tree. Vertices represent states of the requirements model and the edges represent transitions between states. The edges are labeled by the stimulus applied, any responses generated, and the associated requirement identifiers. The root vertex represents the specified initial state for the system. Paths originating at the root represent valid test scenarios for the modeled system. All test scenarios are extracted from the scenario tree by performing a modified depth first traversal. The test scenarios are represented as a sequence of stimulus/response pairs.

1.2.4.1 Use of Experimental Frames

Application of test scenarios to design models is performed through the concept of an *experimental frame* (EF) [29, 30, 31]. The EF is a construct which helps the model builder to keep the design model (the object of interest in this context) separate from the stimuli and responses used for evaluation during simulation. This separation is analogous to the idea of a test bench and unit under test used in hardware simulations.

The separation of the design model and the EF has many benefits. An EF helps to define the aspects of interest for the design model. Since it is not practical for a model to accurately describe every possible behavior that the real world object might exhibit (in our context this is the final design which does not yet exist), a subset of behaviors are usually selected. The EF will generate input parameters and observe output parameters that are related to these behaviors of interest. More than one EF may be used with the same model in order to test different behaviors or to test at different levels of model granularity. As long as the interface for the design model is well defined, alternative design models may be tested with the same EF.

The EF is decomposed into three components, the generator, transducer, and acceptor. The generator is responsible for applying input segments (a series of system stimulus) to the MUT. The transducer is responsible for collecting and verifying system responses and for calculating performance measures. The acceptor interfaces to the operator and monitors the state of the simulation in order to control the startup and termination of the experiment. The coupling of the EF components and the coupling of the EF to the design model is illustrated in Figure 3.

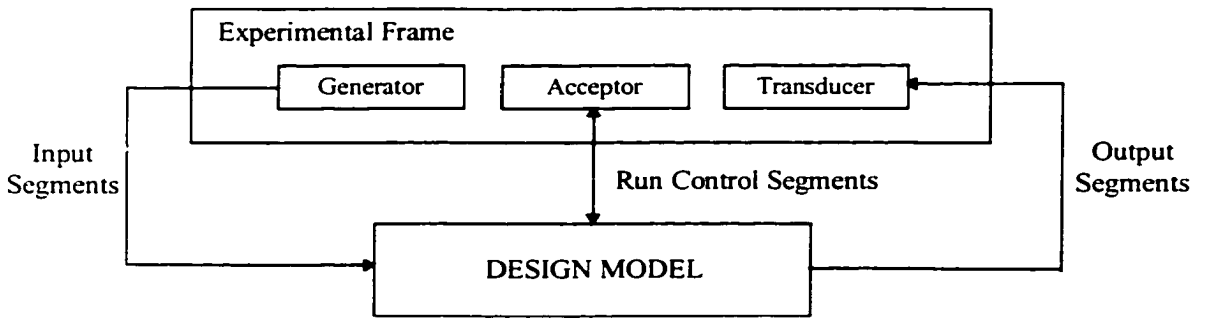


Figure 3. Structure and Coupling of an Experimental Frame

When the design proceeds to the prototype level, the same set of test scenarios will be used and interpreted, possibly after suitable format translation, by a real-time test environment [32, 33].

1.3 Observations on Requirements

This section summarizes observations regarding the current state of practice and research on requirements engineering.

1.3.1 Approaches to Documentation of Requirements

There are many approaches to the documentation of requirements. One approach that is still widely practiced today is that requirements are maintained as natural language text throughout the design process. When used as the only method to document requirements, this is inherently imprecise. It is extremely difficult to identify deficiencies and inconsistencies, particularly for a large set of requirements. Problems with the requirements may become evident only in latter stages of the design process when a certain level of detailed design has been completed and the problems are exposed.

At the other extreme, the requirements may be converted into a language based on formal logic. The formal description of requirements allows for the use of theorem proving methods in order to prove the correctness and consistency of the given set of requirements. A selected summary of such specification languages can be found in [34]. These methods, however, are not widely used in practice due to difficulty in the conversion process and the fact that most designers have not been trained in these formal methods.

There are three goals for the formulation of requirements as described in this work. The first is to be formal enough to allow for automated reasoning related to the documented requirements (e.g., test generation). The second is to maintain the requirements in a format that is easily understood by the design and systems engineers. Finally, of primary importance is to ensure that the proposed formulation of requirements easily builds on existing requirements documentation methods. The first item will provide an improvement to the design process while the second and third will allow it to be used in practice.

1.3.2 Taxonomy of Requirements

During the course of researching requirement methods and from experience in practice, it has become clear that there are many different types of requirements. The first and most obvious distinction is between functional and non-functional requirements. Functional requirements state what the system should and should not do. Non-functional requirements state constraints under which the system must still perform its function. Examples of non-functional requirements are power consumption, weight, and size limits and compatibility such as a requirement to interface to a standard bus or to use a particular processor family to allow software reuse.

Performance requirements state how well the system must function. Recall that behavior is the combination of function and performance. By this definition, the class of requirements called behavioral requirements will contain both functional and performance requirements. Behavioral requirements are the requirements of interest to

this research and may specify the required function or the required performance of a previously specified function.

It has also become apparent that behavioral requirements most often define what the system should do. We call these types of behavioral requirements *positive action* requirements. Conversely, *negative action* requirements define what the system should not do. It must be noted that quite often the negative action requirements are omitted and are implied by the negation of the positive action requirement. This is important when requirements are used for automatic test generation and will be discussed in more detail in a later chapter.

1.4 Observations on Testing

This section will describe some observations regarding testing that have been made during the course of this research.

1.4.1 Testing: Why?, When?, What?

Why do we test? One simple answer is that we test to gain confidence that our designs will function as we expect. Design descriptions, models, and physical systems are inherently static until placed into their target environment and allowed to operate. The type of testing described here is usually referred to as engineering test. This type of testing is usually ad hoc.

We test to demonstrate correctness. When a design is being presented as complete, how can management or the customer be convinced that the design is in fact correct? The type of testing used to answer this question is usually referred to as formal

qualification or formal acceptance testing. It is intended to show that a design meets all requirements. This type of testing may be required by contract or self imposed as part of a good design process.

We test to verify there are no hardware failures. When a system has been designed and is in production, how can the manufacturer be sure that each unit produced is free of hardware defects? The type of testing used to answer this question is usually referred to as production test. This type of testing may not be necessary, as some manufacturers have demonstrated, if the production process is controlled tightly enough. However, production testing remains very common today.

When and what do we test? We test models, modules, components, and prototypes during development. We test prototypes and final products at design completion. We test sub-assemblies and final products during production and during maintenance.

1.4.2 Design vs. Production Testing

The preceding section has touched on a subtlety regarding design vs. production testing. Most engineers are aware of this difference, although possibly at a subconscious level. The difference is the goal of the testing. In design testing the goal is to demonstrate the correctness of the design with respect to the requirements. In production testing the goal is do demonstrate that a particular instance of the system behaves as designed.

Design testing is complicated by the fact that the requirements may still be evolving. A measured behavior may bring to light an issue that has not been adequately addressed in the requirements. This may lead to altering the design, the requirements, or both. In

addition, design testing seeks to demonstrate that the design is correct. This is a goal that may be impossible to attain through testing for systems of any reasonable amount of complexity. However, in order to get as close as possible to this goal, the design tests are made as comprehensive as possible. As a result, the suite of test cases making up a formal qualification test is generally much larger and complex than a suite of tests making up a production test. A final observation on design testing is that tests should be generated without consideration of any particular design. There will always be multiple ways to design a system that meets the same set of requirements and each should pass the same suite of design tests.

Production testing is complicated by a conflicting set of constraints. Since these tests are to be performed a large number of times, usually to be to be executed on each system produced, it is important to minimize the amount of time the test takes in order to reduce cost. At the same time, when a failure is detected, the information gained should be complete enough to easily diagnose the failed component. In contrast to design tests, production tests can and often need to use information about the particular system being tested. An example is digital logic circuit test where the structure of the circuit is used to optimize the set of test vectors to be applied.

1.4.3 Alternative to Design Testing?

If there is any alternative to design testing, it would seem to be formal proofs of correctness. In order to facilitate application of proof techniques to design descriptions, the design descriptions must be captured in a very precise and formal logic specification. This is an active area of research [34, 35, 36, 37] and can provide benefits to the design

process, but it is this author's position that these techniques would never completely remove the need for testing.

The table below provides a partial list of pros and cons for testing and correctness proofs.

Table 1. Correctness Proof Vs. Testing Pros and Cons

Correctness Proofs		Testing	
PROS	CONS	PROS	CONS
Correctness Guaranteed.	Formal specification required, requiring mapping do downstream design descriptions.	Applied at multiple levels of design descriptions which verifies the mappings between levels if the tests are consistent.	Correctness not guaranteed.
Can be applied early.	Can be computationally complex.	Can be applied early.	Can be computationally complex to generate.
May be applied in hierarchical manner.	When a proof fails it may be difficult to ascertain why.	Experimentation improves understanding.	Can be expensive to apply.
	Limited system applicability.	Builds confidence.	Quality of tests may be hard to quantify.
	Stuck at virtual level.	May be required.	May be hard to validate test results.
	Formal specification is an abstraction of actual design.	Tests may be reused (regression testing).	Testing may be difficult due to accessibility at system level.
	In all likelihood, testing will still be necessary.	Tests may be applied to multiple design alternatives.	Cost of test maintenance.
		Tests may be applied to modules or components (hierarchical testing).	

Without attempting to apply a weighting scheme to this partial list of items, it can be noted that the number of CONS for each is equal, but the PROS for testing outnumber the PROS for correctness proofs.

1.4.4 Approaches to Automatic Test Generation

There are many possible approaches when considering how tests might be automatically generated. A sample of some of these will now be discussed.

State-based approaches are probably the most common [9, 10, 18, 38, 39, 40, 41, 42, 43, 44, 45, 55, 57, 61, 62]. These approaches generally explore the state space of a design description and use various criteria to determine when the set of generated tests is complete. The approach presented in this work falls into this category.

Another approach would be to enumerate input combinations [46]. The approach is fairly straight forward but may lead to test sets that are highly redundant and as a result larger and more costly to use than necessary.

Generating tests based on the implementation has previously been mentioned in relation to production testing. While appropriate for production testing, this approach does not seem well suited to design testing. The reason is that tests generated in this manner will help answer the question of whether the system has been build right, but not whether the right system has been built. This is fine for production testing where it is already assumed that the right system has been built. If a design omits a required function, it will not be detected, and likewise, if a design contains an extra function, it will be tested but the fact that it was not required might get overlooked. In the realm of software engineering this approach is usually referred to as code-based testing, and different coverage criteria may be applied such as branch cover or condition cover. These criteria refer to the comprehensiveness of the tests and define whether one or all conditions enabling a branch to be taken are tested.

Tests can also be generated in a random manner. This approach might be applied in situations where the state space is too large to allow practical use of other methods. If used, this approach may not produce the level of confidence gained with tests generated through more precise methods. However, a practical use of this approach is robustness testing that is used in conjunction with a precise set of tests used to qualify a system. After qualification, when the system is believed to meet all requirements, a large set of variations of inputs and modeled variations of physical parameters may be applied to see how robust the design is. This is similar to an accelerated life testing idea where one hopes to expose the system to a large number of the possible conditions the system will experience throughout its expected life span.

There is always an ad-hoc approach. When taken as a whole, the set of engineering tests that may be applied to a system and various sub-systems will appear to be ad-hoc. Even though each test will be developed to verify a certain aspect that is important at the time, unless an overall plan and organization of these tests is enforced, it will be difficult to make any statements about the coverage level the tests have provided. This method would not be used for formal acceptance testing.

Finally, tests may be generated based on requirements. This method is commonly used to prepare for qualification testing and is the basis for the approach to automatic test generation presented in subsequent chapters. However, in practice this approach is usually applied manually. Tests are developed, ideally by a team of test engineers that are independent of the design team, and the tests and requirements are mapped to each

other to assure that all requirements have an associated test. A test might cover multiple requirements and a requirement might be tested by multiple tests.

1.4.5 Benefits of Automatic Test Generation

The primary target for the automation presented in this work is the requirements-based approach to test generation, which is most often applied manually today. If this approach can be applied manually, and apparently with success, then why should it be automated?

The generation of a meaningful and complete set of test cases is a time consuming and tedious task, as this author can attest from experience. There is certainly the usual benefit of automation, namely improved efficiency and in this case relieving the human from the task (perhaps there are those who would enjoy such a task, but it is assumed that most engineers would not). In addition, when the tests are generated manually, there is always a question as to their completeness. Did the test designer allow a conscious or unconscious bias to enter into the tests, which left gaps in the testing that in turn might let a design flaw go undetected? When the tests are generated automatically by a set of algorithms, it should be possible to remove these concerns stemming from human biases. It might be the case that the test generation algorithms contain a flaw, but once detected and corrected, it should no longer be a concern. The same cannot be said of human test designers, where the issues will vary from designer to designer and even a single individual's approach to test generation may vary from system to system.

1.5 Problem Definition and Research Goals

The problem to be solved by this work is to define a process that takes as input the requirements for a system captured in natural language format and produces as output a set of test scenarios that adequately tests the system.

The research goals are numerous. First and foremost is to define a theoretic basis for addressing the stated problem. In the process of defining the basis for solving the problem, restrictions to the problem that allow the problem to be solved will be identified. These restrictions will in turn expose areas that require further research and possibly very different approaches.

From the theoretic basis will come a definition of *adequately*. An adequate set of tests may vary depending on the test objectives. The variables available to control the test generation process need to be identified so that the process can be designed to be controlled by these variables.

Having a theoretic foundation will in turn support another research goal, which is to automate the defined process to the greatest extent possible. As stated previously, manual processes exist and are being used in practice today. In order to facilitate automation, a set of algorithms will be developed for certain steps in the test generation process. Some of these algorithms will be further developed into working prototype tools to better illustrate the viability of the proposed process.

2 APPROACH TO TEST GENERATION

This section contains an overview of how the test generation algorithms generate test scenarios from requirements and the issues faced while developing these algorithms. Related works are cited throughout this discussion.

2.1 Evolution of the Base Test Scenario Generation Approach

Application of state-space techniques seemed to be a good approach to aid in automatic generation of test scenarios. State-space techniques have also been applied by other researchers, the earliest of which was Chow [43].

The initial approach was very simple and can be described as follows. Develop a model of the required functions of the system and then apply inputs to drive the model into new states. The set of unique states that have been reached is recorded so that only state changes resulting in new states are allowed to be expanded through the application of additional input events. This approach will expand the full state-space reachable through application of the given set of inputs.

To support this approach, a set of requirements forms was proposed to capture the needed information. Details of the initial approach to test generation and of the requirements forms are contained in [47]. Development of these forms was not pursued due to time constraints and the fact that a suitable alternative was found in the form of the SCR (Software Cost Reduction) formalism and the supporting tool set [6, 17, 48, 49, 50, 51]. (Section 2.4 describes the use of SCR to support test generation.)

This initial approach had the advantage of being fairly simple to implement, but suffered from the well known state-space explosion problem. While this approach is

practical for smaller systems, the state-space may quickly grow beyond a manageable size for more complex systems. This problem can be lessened through constraints on the application of system inputs, but cannot be removed without a fundamentally different approach to the problem.

The primary purpose of a set of test cases within Model-based Codesign is to validate the proposed system design against the system requirements and to provide measurable test data to support design alternative analysis. In light of these goals and the state explosion problem, it was appropriate to explore the possibility of applying a more "intelligent" test generation approach. Only a sufficient subset of the scenario search tree (i.e., the total state space of the requirements model) should be generated such that when all scenarios from this subset are applied to the system, all system requirements have been covered. In order to accomplish this, the test generation algorithm will have to be modified from the initial approach and will require additional information.

The information needed will be the complete set of requirements and a mapping of these requirements onto the requirements model. The process of developing the requirements model is the conversion of the textual requirements into the formal notation. Every action in the model (i.e., state transition or generation of an output) must exist to fulfill one or more system requirements or parts thereof. An action that fulfills part of a requirement will exist when the textual requirement is compound in nature or, for instance, requires multiple outputs. An example of an action needed by multiple requirements would be if a particular output was defined to be generated by two separate

requirements, possibly in response to different stimuli or in response to the same stimulus but in distinct system modes.

In order to better understand this formulation of the test generation problem, a definition of "covering" is needed. A set of test cases that cover all system requirements is one that causes at least one state transition (possibly generating a system response) associated with each uniquely identified system requirement.

In the initial approach, all test cases defined by the complete scenario search tree represented all possible stimulus and response sequences for both the system and environment (under a one stimulus at a time constraint). The other extreme would be to require each system response to be triggered at least once within the set of scenarios. This would be analogous to software testing with a branch cover criterion. A more robust approach would be to require that actions with disjunctive triggers be separated into individual actions. For example, a single state transition triggered by event A or B and associated with requirements 1 and 2 would be separated into two distinct transitions each associated with a single requirement. This approach will ensure that every action specified in the requirements model will be exercised at least once by any set of test cases that cover all requirements. This is analogous to software testing with a condition cover criterion.

The desired output of the test case generation process is a minimal set of test cases that cover all requirements. Minimal will be defined to mean a set of test cases in which no test case may be removed without destroying the covering property. The final approach to test generation is based on this formulation as a covering problem.

2.1.1 Development of a Solution to the Requirements Covering Formulation

Formulating the problem into a covering problem provides the benefit of a potentially reduced set of test scenarios (i.e., avoidance of the state-space explosion problem), but since nothing is free, what is the cost? The cost is the increased complexity of solving the problem.

2.1.1.1 Requirements Covering is NP-Complete

How hard is the Requirements Covering (RC) problem? This problem is a search problem since a search tree is being generated. However, instead of searching for a particular goal state, the search is for a set of edges that cover all system requirements. Searching, or generating the entire tree is not hard in and of itself. The tree is a concise representation that may be generated (or traversed if the tree already exists) in polynomial time with respect to the number of vertices in the tree, by a depth first traversal for example. So this portion of the problem is not hard. What makes this problem hard is that the actual input is not the vertex (state) set, it is the requirements model which defines a state machine. The size of the tree is exponential in relation to the transitions defined in the model.

If it is assumed for the moment that the scenario search tree has been created, it can also be shown that finding the minimum set of scenarios from such a tree is NP-complete. Minimum will be defined to mean a set of scenarios that collectively contain the fewest system state transitions (edges). In making this definition it is assumed that there is no preference between a large number of short scenarios and a small number of long

scenarios. A reduction from the RC problem to the Set Covering problem, which is a known NP-complete problem [52], is now given.

In order to proceed, a more formal definition of the RC problem is needed. Let an instance be given by (T, R), where T is the rooted scenario search tree and R is the set of all system requirement identifiers. An example is shown in Figure 4. Each edge of T is labeled by a subset of R. The problem is to find a set of rooted paths, where the union of all edge labels is equal to R, and collectively the paths contain the minimum number of edges. This optimization problem can be converted to a decision problem by adding to each instance a goal for the number of edges, k, where k is an integer greater than zero.

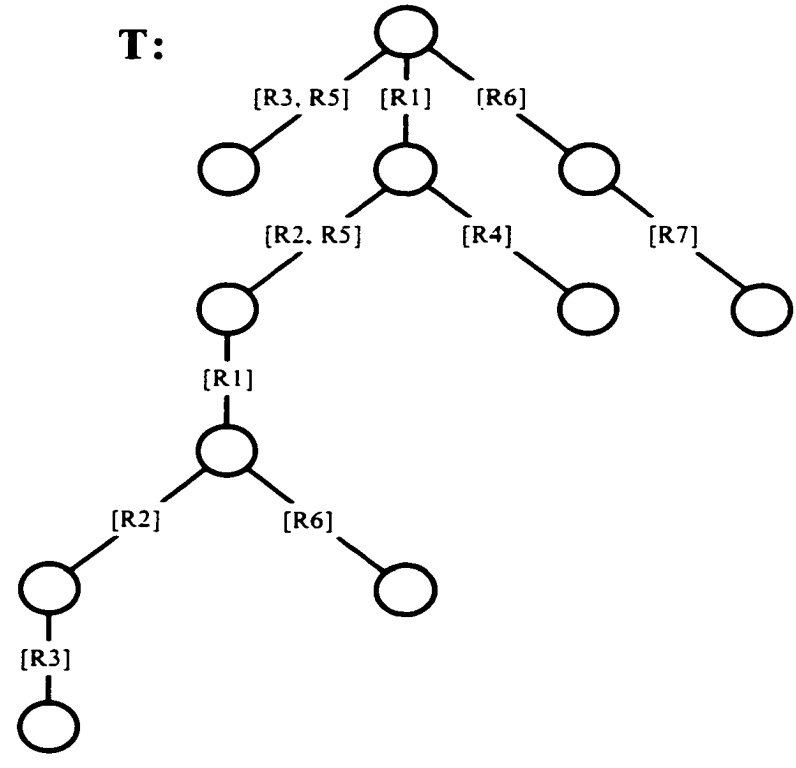


Figure 4. Example Scenario Search Tree

Minimum scenario is in NP: A potential solution may be guessed by selecting arbitrary rooted paths, in the form of an ordered set of edges, from the scenario search tree. These sets can then be checked by verifying that 1) the total number of the edges in all sets is $\leq k$, and 2) the union of the edge labels of the members of all sets is equal to the set of all system requirements, R. The guessing stage and the two tasks for the check stage can be calculated in O(|e|) time each, where |e| is the number of edges in the scenario search tree.

Similar known NP-complete problem: The set covering problem consists of (X, F), where X is a finite set and F is a group of subsets of X such that every element of X is a member of at least one subset in F. The problem is then to find C, the smallest subset of F, where the union of all members of C is equal to X. This optimization problem can be converted to a decision problem by adding to each instance a goal k, where k is an integer greater than zero. The problem then is to find C, a subset of F, such that $|C| \leq k$.

Reduction from minimum scenario to set covering: The set of system requirements, R, is mapped directly to X. The set of subsets, F, can then be generated by traversing T from the root in a depth first manner. Each member of F is created when a new requirement is discovered along each path. For example, the longest path in Figure 4 contains the labels [R1], [R2, R5], [R1], [R2], and [R3]. The resulting subsets to be added to F would be {R1}, {R1,R2,R5}, and {R1,R2,R3,R5}. The complete F for this example is: {{R3,R5}, {R1}, {R1,R4}, {R1,R2,R5}, {R1,R2,R3,R5}, {R1,R2,R5,R6}, {R6}, {R6,R7}}. If an instance of the set covering problem where to be created as

described, the solution produced would not be optimal given the definition of minimum above. The problem is that a solution would contain the fewest number of scenarios rather than a set of scenarios with the fewest system state transitions as desired (i.e., the solution would most likely be biased toward a small number of long scenarios). The reason is that the basic cost function gives a cost of 1 for each set selected from *F*. For the example in Figure 4, the optimal solution using this cost function is {{R1, R4}, {R1, R2, R3, R5}, {R6, R7}}, which contains three sets (scenarios) giving a cost of 3. However, this solution has a cost of 9 in terms of the number of state transitions (i.e., the number of edges or the total test length). In order to obtain the desired result, the cost function must be altered. The correction is to associate a variable cost with each member of *F*. The cost will be the number of transitions (edges) in the rooted path associated with each member of *F* generated as described above. This cost can easily be calculated by keeping track of the depth in the tree during the generation of *F*. For the example in Figure 4, the costs for the member of *F* would be as shown below.

F = {{R3,R5}, {R1}, {R1,R4}, {R1,R2,R5}, {R1,R2,R3,R5}, {R1,R2,R5,R6}, {R6}, {R6,R7}}
Cost = 1 1 2 2 5 4 1 2

An optimal solutions using this revised cost function is {{R3,R5}, {R1,R4}, {R1,R2,R5}, {R6,R7}}. This solution contains four sets (scenarios), but has a cost of only 7 state transitions.

Reduction can be computed in polynomial time: The generation of the members of *F* can be calculated by a depth first traversal of *T*. A depth first traversal has a run time bounded by O(|n| + |e|), where |n| and |e| are the number of vertices and edges in *T*

respectively. Since T is a tree, this run time reduces to O(|n|). The operation at each step of the depth first traversal will be to maintain an ordered list of requirements covered by the present path. Keeping this list ordered will allow searching, followed by a possible insertion (when following an edge forward) or deletion (for following and edge backward), to be performed in log|R| time. Thus the total run time will be O(|n| + log|R|) = O(|n|), since $n \leq R$.

At the start of this proof, the assumption was made that the scenario search tree was available. Since the goal is to avoid generating the complete scenario search tree, our problem is actually more difficult.

2.1.1.2 Heuristic Approach to Solving the Requirements Covering Problem

Now that the hardness of the problem has been demonstrated, it is appropriate to seek an approximation or heuristic approach that will provided an adequate solution in a reasonable amount of time.

In order to propose a suitable heuristic, some further analysis is needed. A good place to start is listing what information is available. The following items are known; the start state of the system, the interface requirements (which define when the environment is allowed to provide stimulus), the system requirements (which have been used to define the state transitions in the requirements model), a list of system requirements to be covered, and a mapping between requirements and the transitions in the requirements model. The requirements mapping allows for the determination of which requirements are covered when a transition is exercised in the model.

Initially, no requirements will have been covered, thus any available stimulus that causes a state change in the model will help reach the goal. This implies that a greedy approach should work well. One possible greedy algorithm would expand all possible states for the most recent state added to the scenario search tree, then add the state connected by the edge covering the most requirements not yet covered. The number of uncovered requirements, which will be called the Requirements Covering Value (RCV), is the quality measure used to select the next state to be added to the scenario search tree. One difficulty is that when a new edge and state are selected and added to the scenario tree, the potential states (states that have been expanded but have not been added to the scenario tree) at that time may have their RCVs change. As a result, the RCV for each potential state will have to be recalculated after each new state is added to the scenario tree. Fortunately, it is not necessary to keep the list of potential states ordered by RCV. During the recalculation pass, simply note the maximum value.

The greedy approach is not a complete solution because what could eventually happen, and is probably quite likely, is that the queue of potential states will at some point contain no entries that are associated with requirements remaining to be covered (and all requirements have not been covered). When this happens, a switch to a second strategy is needed, because the greedy approach would degenerate into a random selection process that in the worst case could expand the entire tree. This situation will occur when there are some requirements that are covered by edges near or at the leaves of the tree. These transitions require that the system be taken through a long series on inputs in order to get the system into a state that allows the needed transition to be taken.

How should the expansion process be guided so that it is likely to follow a path leading to a needed transition?

If the scenario search tree already existed the states connected to a desired transition would be known. Then, for all of the potential states, a distance to the enabling state for the desired transition could be computed. This distance could be the number of state variables that do not match. The selection of the next state to expand could be made based on this measure, and the expansion could then always follow the most promising path so far. Since the scenario search tree is not available to work with, this approach cannot be used directly. What *is* available is the requirements model. A transition that covers one of the elusive requirements can be identified and from the conditions enabling this transition, a state can be defined (most likely with *don't care* values) that will allow the desired transition to be taken. The selection of the next state based on the distance measure can then be applied as described above. If *don't care* values exist in the enabling state, they will not count in the distance calculation. This type of search will be called *Distance-based* search.

During the distance-based search phase of the algorithm, if the edge leading to the next state selected does not cover any additional requirements, it will not be added to the scenario tree. The reason is that there is no guarantee that any scenario containing that transition will eventually lead to the covering of additional requirements. Instead, such states are kept in the list of potential states. After each new state is expanded, all new edges will be checked for additional coverage. If any are found, a backward trace is

initiated that continues until it connects with the current scenario tree. All edges and states along the path defined by the backward trace are then added to the scenario tree.

This dual algorithm approach is described in more detail in section 3.1 and in [53]. The test scenarios defined by the scenario tree resulting from the application of the greedy and distance-based search algorithms will be referred to as the *base scenarios*.

2.2 Issues with the Base Scenarios

In this section, a limitation of the base test scenarios is described, namely that these test scenarios are not suitable for *black box* testing. The reason black box testing is important, related works, and the method used to address this issue are described in the sub-sections that follow.

2.2.1 Are the Base Test Scenarios Complete?

Each test scenario generated forces the system to traverse a set of state transitions. Unfortunately the base test scenarios only require that state transitions are traversed and not that the effect of each transition can be verified by observing some change at a system output. This is important in our context (i.e., Model-Based Codesign) since the test scenarios will be applied to design models and physical prototypes. Both of these targets are to be treated as a black box. The first, since the correspondence between the state variables of the requirements model and the design model will most certainly differ, and the latter, in addition to the state mapping problem, due to the limitations of physical access to internal changes.

The problem will now be illustrated with an example. The example system is a Safety Injection System for a nuclear reactor adapted from [54]. A block diagram of the system is shown in Figure 5. The basic operation of this system is to monitor *WaterPres* and assert *SafetyInjection* to raise the water pressure when *WaterPres* is sensed to be below the predefined threshold, LOW. The *Block* input is used to allow the operator to "block" or override the assertion of the *SafetyInjection* output. *Reset* is used to unblock the system which re-enables normal control of *SafetyInjection*. The input *TRef* (Time Reference) is monitored by the system when the system is blocked. As a safety mechanism, the system will autonomously unblock itself after three events are sensed on *TRef*.

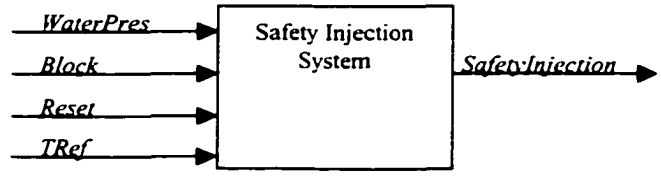


Figure 5. Safety Injection System Block Diagram

The state variables for this system are *Pressure*, *Overridden*, *TrefCnt*, and *SafetyInjection*. *Pressure* is an abstraction of *WaterPres* and is represented by an enumerated type with values of TOOLOW and PERMITTED. The values of TOOLOW and PERMITTED are set based on whether *WaterPres* is above or below the predefined threshold LOW. *Overridden* is a boolean variable which is set when the operator asserts *Block* and reset when the operator asserts *Reset*. *Overridden* will disable *SafetyInjection* even if the *Pressure* indicates that *SafetyInjection* should be set to On. *TRefCnt* is an integer count of the number of events that have occurred on the input *TRef*.

SafetyInjection is enumerated with values of On and Off and adds water to the cooling system, which increases *WaterPres* when set to On. The initial state of the system is specified to be (*Pressure*, *Overridden*, *TRefCnt*, *SafetyInjection*) = (PERMITTED, False, 0, Off).

One of the base scenarios that has been generated for this system is:

WaterPres = 50 / *SafetyInjection* = On

Block = On / *SafetyInjection* = Off

Tref /

Reset = *On* / *SafetyInjection* = On

where the scenario is given in the form *input* / *output*. This scenario takes the system sequentially through the states listed in Table 2.

Table 2. Safety Injection System Sequential States for Example Scenario

Pressure	***Overridden***	***TrefCnt***	***SafetyInjection***
PERMITTED	False	0	Off
TOOLOW	False	0	On
TOOLOW	True	0	Off
TOOLOW	True	1	Off
TOOLOW	False	0	On

The system has a requirement to become "un-blocked" after receiving a *Reset*. This requirement can be immediately verified by observing whether *SafetyInjection* changes to On after *Reset* is applied in the last step of the scenario. The system also has a requirement that the number of events occurring on the *Tref* input be counted so that if

the system is "blocked" while *WaterPressure* is below the specified threshold, the system will automatically "un-block" after the third *Tref* event. Since a condition for the system to count events on *Tref* is that the system is "blocked," after *Reset* "un-blocks" the system, any accumulated count of *Tref* events must be cleared. The fact that this occurs can be noted in the final state where *TrefCnt* is once again set to zero.

When applying this test scenario to a black box system, the fact that *TrefCnt*, or the equivalent state variable in the system under test, was set to zero cannot be verified. This points out that our problem concerns requirements (or parts thereof) that only effect the internal state of the system, which will be called *Internal Change* (IC) requirements. Let test scenarios that do not allow for the verification of all IC requirements associated with that scenario be called "incomplete scenarios." The problem to be solved, then, is that of how to identify and then enhance the incomplete base scenarios so that every requirement considered to be covered by that scenario can be verified by observation of some system output.

2.2.2 Identifying Incomplete Base Scenarios

The approach selected to identify incomplete base scenarios is analogous to methods for automatic test pattern generation for digital systems where the circuit is simulated both with and without the injection of a circuit fault. Inputs that are found to produce different outputs for the fault free and faulty circuits are valid tests for the given fault.

In the problem of identifying incomplete scenarios, the goal is to determine if any system output would differ if the covered requirement was not implemented. The technique described above can be applied where the "injected fault" is to disable the

target requirement. Since the requirement identifiers are mapped onto the state transitions of the requirements model, this is equivalent to disabling a single transition in the model.

Each requirement identifier listed as being covered by each base test scenario must be considered individually. Call the requirement identifier being considered the target requirement. Both the correct requirements model and the model with the transition associated with the target requirement disabled are simulated in parallel while the associated base test scenario is applied to the system. At each step the system outputs for the two models are compared. An observed difference confirms that compliance with the target requirement can be verified by observing the system outputs while applying the existing base test scenario. Of course the simulation may be stopped as soon as a difference is observed. If, on the other hand, the entire base scenario is applied and no system output difference is observed, then that test scenario is incomplete with respect to the target requirement identifier.

2.3 Evolution of the Approach to Enhance the Base Scenarios

After the incomplete base test scenarios are identified, they must be improved or enhanced to support black box testing. This section contains an overview of how the incomplete test scenarios are enhanced and the road that lead to the approach selected. Related works are cited throughout this discussion.

2.3.1 Related Approaches to Black Box Testing

The problem at hand, when in the domain of finite automata and general I/O systems, is termed in the literature as the problem of finding a distinguishing or diagnosing experiment [55, 56, 57, 58]. In the domain of linear systems theory the problem is given the term observability [59]. Since our requirements model is based on the finite state paradigm, linear systems theory where systems are described by continuous functions, will not be discussed further.

Gill's early work [55] defined the concept of experiments with finite state machines. An experiment refers to the process of applying input sequences to a given machine and observing the resulting outputs in order to ascertain unknown attributes. The term diagnosing experiment was used to describe an experiment where the goal is to determine the starting state of the machine. This is more commonly referred to as a distinguishing experiment in more recent literature [56, 57, 58].

Distinguishing experiments may be preset or adaptive. Preset experiments apply a fixed input sequence. Adaptive experiments apply an input sequence that varies depending upon the outputs generated by the machine. In most cases the adaptive experiments will be shorter than preset experiments and in some cases, a machine may have an adaptive experiment without having a preset experiment.

A distinguishing experiment, in its most general sense, will provide a unique output for each possible initial state. Not all finite state machines have distinguishing experiments, even if the machine is minimal. However, a minimal machine cannot have undistinguishable states. This implies that while a minimal machine may not have a

completely general distinguishing experiment capable of identifying all possible machine states, it is always possible to create a distinguishing experiment that can distinguish any pair of states. These are termed pair-wise distinguishing experiments by Gill.

The procedure for generating distinguishing experiments is similar for preset and adaptive experiments. In both cases it is assumed that the transition function of the machine is known. A successor tree is formed, where the root is the set of possible initial machine states. Each level of the successor tree is created by applying all valid inputs to each vertex at the previous level. When a successor state can be distinguished due to a unique output, the set of possible states is split. Each vertex then contains a vector of sets, called an uncertainty vector. If a vertex having a vector where each set contains only one member (a trivial uncertainty vector) has been created, a distinguishing experiment has been found. The sequence of inputs along the path from the root to the vertex with the trivial uncertainty vector is the distinguishing experiment. The tree is altered to show the possible machine responses when generating an adaptive experiment.

An algorithm for generating a successor tree for distinguishing experiments was given by Gill [55] and later by Kovahi [56]. Deshmukh and Hawat [57] provide a tool implementing the algorithm (the tool also generates homing and synchronizing sequences).

Other related work can be found in [60] and [61]. These works provide examples of test generation for fault diagnosis. A fault (there are many fault models) refers to the situation where the implementation of the finite automata is not operating as designed. A key point that must be emphasized here is that the goal of test generation in our case is to

validate the design against the system requirements. This is different from fault diagnosis, where it is assumed that without the presence of a fault, the automata will operate as expected. The primary difference is that in fault diagnosis, the internal structure of the implementation is required for test generation and defines the number of possible faults that must be covered. In our case, it is assumed that the automata is fault free and we have no knowledge of the internal structure. Therefore, test generation for fault diagnosis makes differing assumptions and is based on a different goal. For these reasons the approaches from this domain of automatic test generation are not applicable to the problem at hand.

Work related to design validation is offered by Chow [43] and Tan et al. [62]. Chow coined the term "characterization set" which is a set consisting of input sequences that can distinguish between every pair of states in a minimal automaton. The characterization set can be constructed by repeatedly performing Gill's algorithm with the initial uncertainty vector set to a different pair of machine states for each execution. Chow uses the sequences in the characterization set as part of his test sequences. Test sequences consist of the cross product of a set of sequences sufficient to force the automaton into all possible states with an enhanced version of the characterization set. The enhancement is due to his assumption that the automaton may not be minimal.

Recently, Tan et al., make practical improvements to the generation of test sequences for design validation. Their domain is implementation of communication protocols with the goal of testing to assure compatibility with other implementations using the same protocol. This has been termed protocol conformance testing. Given that the procedure

proposed by Chow can be viewed as somewhat of a "shotgun" approach that would generate an unreasonable number of test sequences for an automaton of more than a few states, Tan describes the difficulty of generating a minimal set of test sequences. He then applies heuristic algorithms to generate harmonized state identification (HSI) sets. The HSI sets are subsets of the characterization set and are used to reduce the length of the generated test sequences.

2.3.2 Solution Approaches

Although the works of Chow and Tan are directed to the same goal (i.e., generation of tests for design validation) the initial assumptions are problematic. In order to generate a distinguishing sequence, it is assumed that the transition and output functions of the automaton are readily available. Although our requirements model is available for simulation and the state variables are known, characterization of the transition and output functions in a form amenable to algorithmic processing is not.

From the forgoing survey and the understanding of our context, a number of approaches to solve the problem at hand seem reasonable. Each of these approaches will be discussed in the sections that follow. A suitable solution should be of limited computational complexity and should be easily integrated with our existing scenario generation algorithms, which use a graph theoretic representation of the system as a finite automaton.

2.3.2.1 Use of Characterization Sets

In order to be able to apply the approach of Chow or Tan, all possible inputs and states need to be identified. Then the next states and outputs generated by applying all possible inputs to all possible states need to be characterized. Since the requirements model is available for simulation, determination of these attributes should be possible. As part of the requirement modeling effort, the state variables are identified and the set of test inputs is determined by manually specifying a subset of values for inputs with a large domain of possible values. This information can be used to limit the characterization effort in the same way they are used to limit test generation. In a previously proposed test generation algorithm [47], all available inputs where applied starting from the specified initial system state in order to generate all possible states, subject to the limitations on the inputs. This algorithm could be applied and the resulting scenario search tree used to characterize the requirements model transition and output functions in a manner consistent with the test generation scheme. After this characterization, the approach of Tan could be used to augment the previously generated test sequences where needed.

The drawback to this approach is that the state space of models grows exponentially with the number of state variables. This will put a strain on computational resources and limit the size of models that can be handled in practice. In fact, the primary reason for developing and proposing the present test scenario generation algorithm [53] based on a requirements cover as opposed to the complete state space exploration approach of our previous algorithm [47] is to avoid the state explosion problem. In addition, while the

SCR requirements model is based on finite state machines, the use of additional "memory variables" called terms and mathematical calculations define a more general model of computation (based on Turing machines). In order to be useful to a wide variety of application domains, the test generation techniques proposed should be applicable to models beyond pure finite state machines. For these reasons, this approach was not pursued.

2.3.2.2 Identify and Target Dependent Requirements

This approach does not appear to have an analog in the literature. The assumptions are that every transition in the requirements model is needed (traceable to a system requirement) and that every transition must affect at least one system output. These assumptions are equivalent to saying that the requirements model is minimal. The first assumption will be verified during the assignment of requirements to transitions. If a transition cannot be related to some system requirement, then either the transition is not needed and should be removed from the model or the requirements need to be revised. If the second assumption is not true, then again the transition is not needed. If a transition does not effect some system output, then the transition performs a function that does not contribute to the observable behavior of the system, assuming a black box model, and it may be removed.

With these assumptions, a scheme to verify the tail state of each test scenario would work as follows. Assume transition T is associated with an IC requirement (i.e., it does not directly cause a system output change). The dependency on the state change caused by T can be traced to at least one dependent output, call it O_d. In the case of multiple

dependent outputs, a reasonable selection criterion would be to select the output arrived at through the shortest dependency list. The transition that finally changes O_d will also be tied to a system requirement, call this R_o. R_o could then be identified with T as a dependent requirement. Any transition directly effecting a system output would have zero dependent requirements.

Now the original test sequences (without tail verification sequences) must be analyzed. Each scenario will be associated with a list of covered requirements. For each requirement, r_i, in the list, the remaining requirements, r_j where $i < j$, must be checked as to whether all dependent requirements of r_i have been covered. If not, then these test scenarios must be augmented with additional inputs to cover the dependent requirements.

Since the original set of test scenarios will cover all requirements it might seem possible to use sequences within this set to augment the scenarios needing tail state verification. However, the original test scenarios assume that the system is in the specified initial state at the start of the test. Since this assumption is not guaranteed to be satisfied, these sequences cannot be used directly.

Instead, the distance-based search algorithm (the second part of the test scenario generation algorithm for the base scenarios) could be used. This algorithm takes a starting state, a set of test inputs, and a target requirement. The state space is explored by applying the test inputs and evaluating all potential states against the state that enables the transition of the target requirement. The state with the minimum distance is selected for expansion at each step. This algorithm would be applied starting from the state arrived at

through the transition associated with the IC requirement and would target the enabling state for that transition's dependent requirement.

While this approach may seem reasonable, further analysis revealed that there are a number of problems associated with it. One of these is that a simple dependency analysis will identify dependencies between model variables, but not between requirements (i.e., state transitions). The result is that while it is easy to determine that requirement R_x alters the value of model variable X, and that system output O depends on X, it is not easy to identify which requirement effects R_o. The reason is that, in general, a number of requirements may alter the value of O. A dependency analysis carried out on the level necessary to identify dependencies of R_o would have to interpret the logical constructs and possible variable assignments leading to an output change. Analysis at this level would solve the original problem since the sequence of needed inputs would also be identified making the goal of identifying a dependent requirement a moot point.

Even if the issues just identified where resolved and the dependent requirements could be easily determined, the distance-based search would not be guaranteed to generate an input sequence that would verify the IC requirement. As an example, return to the scenario from section 2.2.1. It was previously determined that the IC requirement, [R4a][1], causing *TrefCnt* to be set to zero at the end of this test scenario was not verified. From an analysis of the system it can be determined that the dependent requirement for

---

[1] The [alpha-numeric] is a label used to uniquely identify a requirement as allocated to a transition in the requirements model. The actual value can be considered arbitrary for this discussion.

[R4a] is [R1]. This determination was arrived at by synthesizing the sequence shown in Table 3, which starts at the final state of the incomplete scenario.

Table 3. Scenario Enhancement Verifying [R4a]

Current State	Input / Output	Next State
TOOLOW, False, 0, On	*Block* = On / *SafetyInjection* = Off	TOOLOW, True, 0, Off
TOOLOW, True, 0, Off	*Tref* /	TOOLOW, True, 1, Off
TOOLOW, True, 1, Off	*Tref* /	TOOLOW, True, 2, Off
TOOLOW, True, 2, Off	*Tref* / *SafetyInjection* = On	TOOLOW, False, 3, On

When the final *Tref* is applied to the system, *SafetyInjection* is set to On through the transition associated with [R1] as desired. Unfortunately, in this system there are multiple ways to enable the transition of [R1]. The [R1] transition is enabled by any state where *Pressure* is TOOLOW, *Overridden* is False and *SafetyInjection* is Off. The distance-based search will use the distance to this enabling state for the target requirement as a guide when selecting the next state for expansion until the transition associated with the target requirement is executed or until no more states are available for expansion. This algorithm would identify the sequence shown in Table 4, which also ends with the *SafetyInjection* being set to On through [R1].

Table 4. Scenario Enhancement that Does Not Verify [R4a]

Current State	Input / Output	Next State
TOOLOW, False, 0, On	*Block* = On / *SafetyInjection* = Off	TOOLOW, True, 0, Off
TOOLOW, True, 0, Off	*Reset* = On / *SafetyInjection* = On	TOOLOW, False, 0, On

Even though the desired dependent requirement was exercised, the original requirement [R4a] has not been verified. The reason is that the system contains multiple

dependencies that fan in and enable the same requirement. This counter example proves that following an incomplete scenario with a sequence that exercises a dependent requirement is necessary but not sufficient to guarantee verification of the IC requirement.

2.3.2.3 The Chosen Approach: Difference-Based Search

Starting with the premise that all IC requirements will cause a state change in the system and that this state change is distinguishable from all other states in the system (i.e., the system is minimal), then there must exist an input sequence that will distinguish the state reached after any IC requirement transition has caused a state change, from any other state. From the incomplete scenarios, the state following the application of an input that exercised the IC requirement, R_{IC}, will represent the state which we need to distinguish. This state will be called S_a. If the "other" state is selected such that the only difference between it and S_a is the effect of R_{IC} then it should be possible to perform a state exploration that is guided by propagating the difference until a difference can be observed at a system output. This type of search will be termed a *difference-based* search and will operate by simulating pairs of states and selecting state pairs that differ for continued expansion. This approach was selected to generate enhancements to the base test scenarios in order to enable black box testing.

2.4 Combining the Base Test Scenarios and Scenario Enhancements

When enhancements have been generated for the incomplete base scenarios, these must be combined with the base scenarios to produce a complete and consistent set of test

scenarios. This process is fairly straight forward, putting aside issues such as differences between state identifiers used in the base scenarios and the enhancements and handling equivalent states, except that in order to avoid inducing redundancy, overlapping scenarios must be identified. Because the scenario enhancements are generated without knowledge of the base scenarios, it is certainly possible that sequences between the base scenarios and the scenario enhancements and even between the enhancements themselves may overlap.

The scenario combining process will first reconstruct the scenario tree representing the base scenarios. Each enhancement will then be processed by identifying the state (node in the scenario tree) at which the enhancement is to be applied and adding the states (nodes) and transitions (edges) corresponding to the steps in the enhancement to the scenario tree. Before a new state and transition are added to the scenario tree, a check for state uniqueness will be made. If the state is not unique, a check will be made to determine if the enhancement overlaps with an existing scenario as defined by the present structure of the scenario tree. If an overlapping condition is identified, the existing state and transition will be used. In this way the scenario tree will represent a non-redundant set of scenarios.

After all enhancements have been added to the scenario, the complete set of scenarios will be output by performing a depth first traversal, as was done to output the base scenarios.

2.5 Use of SCR

The SCR (Software Cost Reduction) method and tool set have been developed at the Naval Research Labs with the goal of improving software quality and to reduce the cost of software development [6, 17, 48, 49, 50, 51]. The SCR method involves creation of a formal state-based model of the system to be developed. This model, which in the current context is referred to as the requirements model, represents a formal specification for the proposed system. The SCR tool set allows for the creation of and contains several automatic techniques for detecting errors in the requirements model. These automatic checks include syntax and completeness, disjointness (non-determinism is not allowed), coverage (state dependent assignments are defined for all appropriate states), state reachability, and cycle detection (mutual dependence which could lead to an infinite loop of state changes). The tool set also includes a dependency graph browser, which allows the developer to visually inspect the dependencies in the model, and a simulator to allow symbolic simulation of the model.

As mentioned in section 2.1.1.2, the information needed to support test scenario generation is the start state of the system, the interface requirements, the system requirements, a list of system requirements to be covered, and a mapping between requirements and the transitions in the requirements model. The SCR models capture all of these except the interface requirements and the list and mapping of the system requirements. The interface requirements are not treated by the present test generation method. This may not be a detriment since, depending on the criticality of the system, it may be desirable or necessary to verify the behavior of the system in the presence of

interface requirements violations. The capture of requirements identifiers and the mapping onto state transitions is easily added to the SCR models by including this information in the Description or Notes free-form text blocks. In addition, the SCR models are open in that they are stored in an ASCII file with a well defined syntax and grammar.

Although in theory, any state-based model could be used to represent the requirements model and be feed into the test scenario generation method, the SCR models include facilities to capture the additional information needed. For these reasons the custom forms initial proposed [47] where not developed and the SCR method and tool set were selected to support the development of the requirements models.

3 SCENARIO GENERATION ALGORITHMS

Scenario generation, as defined by this work, is a four part process. The first step is to generate sequences of inputs that exercise (cover) all of the identified requirements. The heuristic approach taken to accomplish this task has been discussed at an abstract level in the previous chapter. The remaining three steps of the process are needed in order to support black box testing. The second step identifies which requirements cannot be verified by observing system outputs. The third step attempts to generate additional sequences of inputs so that all requirements can be verified through observation of system outputs. The final step combines the scenarios from steps one and three in order to form the final set of test scenarios. The remaining sections in this chapter describe the algorithms used to perform each step of the test scenario generation process in detail. Although the correctness of these algorithms have not been formally proven, correctness arguments are provided.

The scenarios generated by this process are sequences of stimuli (inputs) to be applied the system under test and the expected responses (outputs) from the system under test. There is no temporal information, other than an ordering, associated with these sequences. The temporal requirements are brought to bear when the test scenarios are converted into the IEEE standard C/ATLAS test specifications [63, 64], which is described in Chapter 5.

3.1 Generation of Base Test Scenarios - The Part 1 Algorithm

The heuristic algorithm to generate the base test scenarios uses the two phase approach previously described: greedy search followed by a distance-based search if

needed. In the following paragraphs this two phase algorithm is presented in pseudo code form. Some details have been omitted in the hope of improving the ease of comprehension. As noted previously, vertices represent states of the requirements model. As a result, references to states and sets of states also imply the corresponding vertices where appropriate.

The primary data structures used by the Part 1 algorithm are defined and initialized as shown in Figure 6.

```
/*****************************************************************
 * SST = Graph representing the scenario search tree             *
 * ST  = Graph representing the scenario tree                    *
 * POT = List of potential states (state not part of the MST)    *
 * RS  = List of requirement identifiers remaining to be covered *
 * COV = List of covered requirements identifiers                *
 *****************************************************************/

Init()
{
  SST = graph with single initial state vertex
  ST  = Null graph
  POT = Empty list
  RS  = Set of all requirement identifiers
  COV = Empty list
}
```

Figure 6. Part 1 Algorithm Initialization

The top level structure of the algorithm is given below in Figure 7. The greedy search phase will continue until no further progress toward covering additional requirements can be made. The potential state set is then cleansed of any states that already exist in the scenario tree since these have already been expanded. If the greedy search has not covered all requirements and there are potential states available, a target requirement is then selected for the distance-based search phase. Finally, all test cases

are output. Note that although not explicitly shown, if at any time during the distance-based search, *POT* is found to be empty and all requirements have not been covered, an error message would be output indicating an ill formed model.

```
nextS = Initial state;
nextE = NULL

do {
  GreedySearch()
} until ( nextS == NULL )

for all s_i ∈ POT
  if si ∈ States(ST) delete s_i from POT

if (RS is not empty and POT is not empty)
  Select r_target from RS and determine its enabling state s_en
else
  r_target = NULL

while (r_target != NULL ) {
  DistanceBasedSearch()
}

Output all rooted paths from ST
```

Figure 7. Part 1 Algorithm Top Level Structure

In the description of the greedy search given in Figure 8, the Expand() function applies all available stimuli based on the given state in order to generate new potential states. Expand() returns the set of new states and their connecting edges so that they may be added to the scenario search tree (*SST*). The function Req() returns the list of all requirements associated with the given edge. ComputeRCV() calculates the RCV for all elements of the given state set and returns the maximum RCV along with the state and edge associated with the maximum.

Greedy search begins with the initial state of the system. Because states already part of the scenario tree are not allow to be expanded and the set of states is finite, the algorithm is guaranteed to terminate. ComputeRCV() returns a covering value relative to the requirement IDs remaining in *RS*. This results in at least one requirement ID being moved from *RS* to *COV* for each state evaluated. Greedy search completes when ComputeRCV() returns a zero value for RCV_{max}, which occurs if *RS* becomes empty or if no requirement IDs associated with transitions leading to states in *POT* match any of the IDs remaining in *RS*. Thus, greedy search will produce a rooted scenario tree where every edge defines a transition that covers at least one requirement ID not covered by any other edge in the tree and will terminate when all requirements are covered or no transition from a state in the scenario tree covers any of the remaining requirement IDs. This argument assumes that Expand() performs a proper simulation of the requirements model so that the scenario tree contains scenarios that are correct for the system.

```
GreedySearch()
{
  if ( nextS ∉ States(ST) )
  {
    (S_new, E_new) = Expand(nextS)
    Add (S_new, E_new) to SST and add S_new to POT
  }
  Add (nextS, nextE) to ST
  Move all Req(nextE) from RS to COV
  RCV_max, s_max, e_max = ComputeRCV(POT)
  if (RCV_max > 0)
    nextS, nextE = s_max, e_max
  else
    nextS = NULL
}
```

Figure 8. Part 1 Algorithm Greedy Search

In the description of the distance-based search given in Figure 9, the ComputeDist() function calculates the distances from each member of the given state set to the enabling state for the given target requirement and returns the state associated with the minimum value found. The BackTrace() function adds states and edges to the scenario tree by performing the backward trace as previously described.

```
DistanceBasedSearch()
{
  s_min = ComputeDist(POT, r_target)
  S_new, E_new = Expand(s_min)
  Add S_new, E_new to SST and add S_new to POT
  for all (s_i, e_i) ∈ (s_new, e_new)
  {
    if any member of Req(e_i) ∈ RS
    {
      BackTrace(s_i)
      Move all Req(e_i) from RS to COV
    }
    for all s_i ∈ POT
      if s_i ∈ States(ST) Delete s_i from POT
    if ( RS is empty or POT is empty )
      r_target = NULL
    elseif ( r_target ∉ RS )
      Select a new r_target from RS
  }
}
```

Figure 9. Part 1 Algorithm Distance-based Search

Distance-based search begins with the set of unique potential states provided by the greedy search. Because redundant states (i.e., states that already exist in the scenario tree) achieved after expansion are removed from *POT* and the number of states is finite, the algorithm is guaranteed to terminate. Of course this assumes that ComputeDist() will return some value for s_{min} as long as *POT* is not empty. Only paths that end in a transition covering a requirement ID in *RS* are added to the scenario tree by the

BackTrace() function. Distance-based search completes when *RS* or *POT* become empty. The former condition is the desired result and the latter condition can occur for an ill-formed requirements model or if inadequate test input values are provided to the Expand() function. Thus, this algorithm will add paths to the scenario tree that cover at least one requirement ID remaining in *RS* and will terminate when *RS* is empty or when the complete state space of the requirements model has been explored. Again, the correctness of the scenarios embodied in the scenario tree is assumed through accurate simulation of the requirements model by the Expand() function.

3.2 Identification of Incomplete Scenarios - The Part 2 Algorithm

After the part 1 algorithm produces a set of test scenarios, these scenarios must be processed to determine which requirements can be verified and which cannot be verified at the black box level. This may be accomplished by simulating each scenario with the effects of each R_{IC} disabled. If the outputs produced by the scenario are not effected, then that R_{IC} has not been verified. It must be noted that the determination of unverified requirements and the application of the difference-based search cannot be accomplished in a single pass over the scenarios. The reason is that the distance-based search may be unnecessarily applied if a single pass is attempted. If the scenarios are processed in a linear manner, and a requirement is found to be unverifiable in the present scenario, and then the distance-based search is applied immediately, it may be a wasted effort since a scenario yet to be processed may already verify the requirement in question. Thus, one pass over all scenarios is required to determine which requirements have not been verified by any of the scenarios. The following algorithm will accomplish the desired

task. The requirements remaining in the *Unverified List* at completion need to be subjected to the difference-based search (the part 3 algorithm). The top level structure of the part 2 algorithm is presented in Figure 10 below.

```
UnverifiedList = List of all requirement identifiers

while (UnverifiedList is not empty and more scenarios)
{
  Select the next scenario, SC_i.
  for each step in SC_i
  {
    for all R_IC ∈ UnverifiedList associated with the step
    {
      Simulate as defined in SC_i from step to the end of SC_i with R_IC
      disabled.
      Stop the simulation if a difference at a system output is
      observed.
      if (an output difference was observed)
        Remove R_IC from UnverifiedList.
    }
  }
}

Output UnverifiedList.
```

Figure 10. Part 2 Algorithm Top Level Structure

The part 2 algorithm starts with the list of all requirements IDs in *UnverifiedList*. At each step of each scenario, the list of covered requirement IDs is compared to *UnverifiedList* and for each ID remaining in *UnverifiedList*, the scenario is simulated with and without R_{IC} disabled. A difference at a system output causes the removal of R_{IC} from *UnverifiedList*. Because there are a finite set of base scenarios and a finite set of requirement IDs, this algorithm is guaranteed to terminate. Termination occurs when *UnverifiedList* becomes empty (the ideal case) or when all scenarios have been processed. Because, requirement IDs are only removed from *UnverifiedList* when an

output difference is detected, the IDs remaining in *UnverifiedList* at termination are those that cannot be verified at the black box level.

3.3 Generating Scenario Enhancements - The Part 3 Algorithm

The incomplete test scenario will provide the state prior to the effect of R_{IC}, call it S_p, and the state after the effect of R_{IC} is S_a. However, since the input I_x applied between S_p and S_a may have also exercised other requirements, S_p and S_a will in general differ by more than the effect of R_{IC}. In order to derive a state equivalent to S_a minus the effect of R_{IC}, call it S'_a, the input I_x can be applied to S_p with the transition associated with R_{IC} disabled in the model. Enhancing the model to allow for transitions to be disabled when desired is handled by special instrumentation of the model function code.

Given the two starting states, S_a and S'_a, the difference-based search would proceed per the high level algorithm description given in Figure 11 below. In this description, the function Expand() will produce a set of child state pairs by applying all available inputs to the pair of starting states.

The difference-based search begins with the pair of states resulting from the simulation of a partial base scenario as described above. Because only state pairs that have not previously been explored are added to the potential list and the number of states is finite, this algorithm is guaranteed to terminate. If the initial pair differs at a system output, then a scenario enhancement of length zero is returned. Note that this check is for completeness and should not occur because such a requirement would have been identified as verifiable at black box level by the part 2 algorithm. If the initial pair of states are equivalent, then an error message is returned.

```
if (S_a ≠ S'_a at some system output)
{
  S_cur  = S_a
  done = TRUE
}
else if (S_a ≠ S'_a)
{
  Insert the pair (S_a, S'_a) onto PotentialList.
  done = FALSE
}
else
  Output an error message and exit

while (not done and PotentialList not empty)
{
  Remove a pair, (S_cur, S'_cur), from front of PotentialList.
  (S_i, S'_i) = Expand(S_cur, S'_cur)
  for each child state pair
  {
    if (S_cur ≠ S'_cur at some system output)
      done = TRUE
    else if (S_i ≠ S'_i and (S_i, S'_i) not previously reached)
      Append (S_i, S'_i) to Potential List.
  }
}

If (not done)
  Output error message.
else
  Perform backtrace from S_cur and return sequence of input/output
  pairs from S_a to S_cur.
```

Figure 11. Part 3 Algorithm Top Level Structure

Because the distance-based algorithm works by propagating a difference, an initial difference must exist. If the initial pair of states differ, but not at a system output, then the search process begins. The state pairs are expanded in a breadth first manner due to the fact the states to be expanded are pulled from the front of *PotentialList* and new states are appended. The search will terminate when a difference at a system output is detected or when the set of potential states becomes empty. The latter can happen when the requirements model is not minimal. Thus, the distance-based search will output an error

if either the initial states are equivalent or the search fails otherwise it will output a scenario (enhancement) that will allow the failure to properly implement the target requirement to be detected at the black box level.

3.4 Combining the Enhancements and Base Scenarios - The Part 4 Algorithm

This algorithm will take as input the base scenarios and the scenario enhancements, which are the outputs of the part 1 and part 3 algorithms respectively. The part 4 algorithm begins by reconstructing the scenario tree defining the base scenarios. This tree is then altered to include the scenario enhancements. The Part 4 algorithm top level structure is presented in Figure 12 and the supporting functions are presented in Figure 13.

The ProcessBaseStates() and ProcessBaseScenarios() functions reconstruct the base scenario tree. ProcessScenarioEnhancements() begins to add each enhancement to the scenario tree by locating the first state of the enhancement in the current scenario tree. As each step of the enhancement is processed, a check is made to determine if that state already exists in the scenario tree. If the state exists, a check is made to determine if it is a child of the state to which the next enhancement state is to be added. This detects situations where there enhancements overlap with existing scenarios and avoids adding redundancy to the final scenario tree.

```
ST = Null graph

ProcessBaseStates()
ProcessBaseScenarios()
ProcessScenarioEnhancements()

Perform depth first traversal of ST and output all scenarios
```

Figure 12. Part 4 Algorithm Top Level Structure

```
ProcessBaseStates()
{
  Add a vertex to ST for each unique base state.
}

ProcessBaseScenarios()
{
  Add edges to ST for each step in all base scenarios.
}

ProcessScenarioEnhancements()
{
  for each scenario enhancement, SE_i
  {
    Locate the starting state vertex for SE_i in ST.
    for each step in SE_i
    {
      Determine if the next state is unique (not in ST).
      if (not unique)
        Detect overlapping paths by determining if the equivalent
        state is a child of the current state.
      if (next state is unique or paths are not overlapping)
        Add new node and edge to ST corresponding to the current
        enhancement step.
    }
  }
}
```

Figure 13. Part 4 Algorithm Supporting Functions

4 TEST SCENARIO GENERATION: AN EXAMPLE

Now that the approach to automatic test scenario generation and the algorithms have been described, an illustration of this process with a small example is in order. The example system is the Safety Injection System briefly described in paragraph 2.2.1. The block diagram is repeated here for ease of reference.

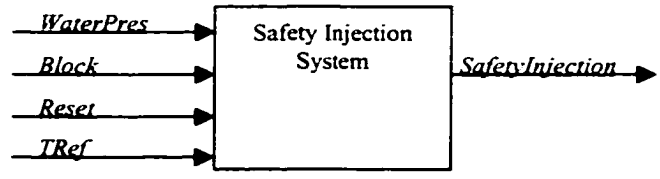


Figure 14. Safety Injection System Block Diagram

Text based requirements for this system are given below. These requirements have been labeled with requirement identifiers. LOW is the pressure threshold which is defined to be 100.

Table 5. Text Based Requirements for Safety Injection System

[R1]	The system shall assert *SafetyInjection* when *WaterPres* falls below LOW.
[R2]	The system shall be considered blocked in response to *Block* being asserted while *Reset* is not asserted and *WaterPres* is below LOW, and shall remain blocked until either *Reset* is asserted or *WaterPres* crosses LOW from a larger to smaller value.
[R3]	Once *SafetyInjection* is asserted, it shall remain asserted until the system becomes blocked or *WaterPres* becomes greater than or equal to LOW.
[R4]	When the system is blocked and *WaterPres* is less than LOW, the system shall automatically unblock itself after the third timing reference event is sensed on input *TRef*.

The requirements model will be illustrated in the Software Cost Reduction (SCR) formalism [48]. As discussed earlier, this formalism is based on finite state machines

specified through the use of tables. In these tables the formula @T(A) is defined to mean the event that A has become true and a variable in *primed* notation (e.g., A') indicates the value that A will assume in the new state. The four tables below define the functions that control the values of *Pressure*, *Overridden*, *TRefCnt*, and *SafetyInjection*.

Table 6. Mode Transition Table for Pressure

Old Mode	Event	New Mode	
TooLow	@T(*WaterPres* ≥ LOW)	Permitted	[P1]
Permitted	@T(*WaterPres* < LOW)	TooLow	[P2]

Table 7. Event Table for Overridden

Mode	Events	
TooLow	@C(*Block*) When (*Reset* = Off) [R2a]	@T(Inmode) [R2b]
TooLow	@T(False)	@T(*Reset*=On) [R2c]
TooLow	@T(False)	@C(Tref) When (TrefCnt = 2) [R4d]
Overridden'	True	False

Table 8. Event Table for TRefCnt

Mode	Events	
TooLow	@C(*TRef*) When (Overridden) [R4a]	@T(False)
TooLow	@T(False)	@C(*Block*) When (*Reset* = Off) [R4c]
TRefCnt'	TRefCnt + 1	0

Table 9. Condition Table for Safety Injection

Mode	Conditions	
Permitted	True [R3a]	False
TooLow	Overridden [R3b]	NOT Overridden [R1]
SafetyInjection	Off	On

These tables have been modified from the standard SCR notation in that requirement identifiers have been added to show the association to the original requirements. The requirement identifier information is actually stored in the free form description block associated with each table provided by the SCR tools. Where the original requirements where compound, the individual requirements have been identified by the addition of an alphabetic character. The P1 and P2 identifiers are a short hand for the fact that these transitions are needed to fulfill multiple requirements. For example, the state *Permitted*, and therefore the transition from *TooLow* to *Permitted*, is needed by requirements R2a, R2b, R2c, and R3a. Appendix A contains the actual SCR model in the native SCRTool SSL format.

The state of the system is defined by: *Pressure* $\in$ {P, TL}, *Overridden* $\in$ {T, F}, *TRefCnt* $\in$ {0..3}, *SafetyInjection* $\in$ {On, Off}. The initial state of the system is (P, F, 0, Off). Note that *Overridden* is symantically equivalent to *Blocked* in the textual requirements.

4.1 Results of the Part 1 Algorithm

Execution of the part 1 algorithm will now be described. Test stimuli for *Reset* is set to On or Off and *WaterPresure* is set to 50 or 150. *Block* and *TRef* are event types and therefore do not require defined test values. The description of the part 1 algorithm will be decomposed into the greedy and distance-based search portions.

Execution of the greedy portion of the part 1 algorithm results in the creation of the scenario search tree illustrated in Figure 15. Edges are labeled by *input/output* and

covered requirements, which are listed in square brackets. The vertex numbering indicates the state name as defined by the set of unique system states listed in Table 10. Note that states 8, 10 and 11 are equivalent to states 1, 3, and 6 respectively and are therefore not part of the set of unique states.

Table 10. Unique System States at Completion of Part 1 Greedy

Name	Pressure	Overridden	TrefCnt	SafetyInj
1	PERMITTED	FALSE	0	OFF
2	PERMITTED	TRUE	0	OFF
3	TOOLOW	TRUE	0	OFF
4	TOOLOW	TRUE	1	OFF
5	TOOLOW	TRUE	2	OFF
6	TOOLOW	FALSE	0	ON
12	PERMITTED	TRUE	1	OFF
16	TOOLOW	FALSE	1	ON

The greedy search always selects the potential state connected by the edge covering the most remaining requirements, and selecting at random in the case of a tie. The greedy portion of the part 1 algorithm continues in this manner until the scenario search tree, *SST*, consists of the states shown in Figure 15. The scenario tree, *ST*, which represents the scenarios, is defined by the bold edges. The edges and vertices that are not part of the *ST* are potential states that did not cover any additional requirements.

At this point the greedy portion of the algorithm will terminate since transitioning the system to any of the potential states covers no additional requirements. The greedy search covered all but two requirements, R4d and R2b.

The distance-based portion of the algorithm will select one of these as a target. Targeting R2b, the state required to enable the associated transition is P, T, X, X, where

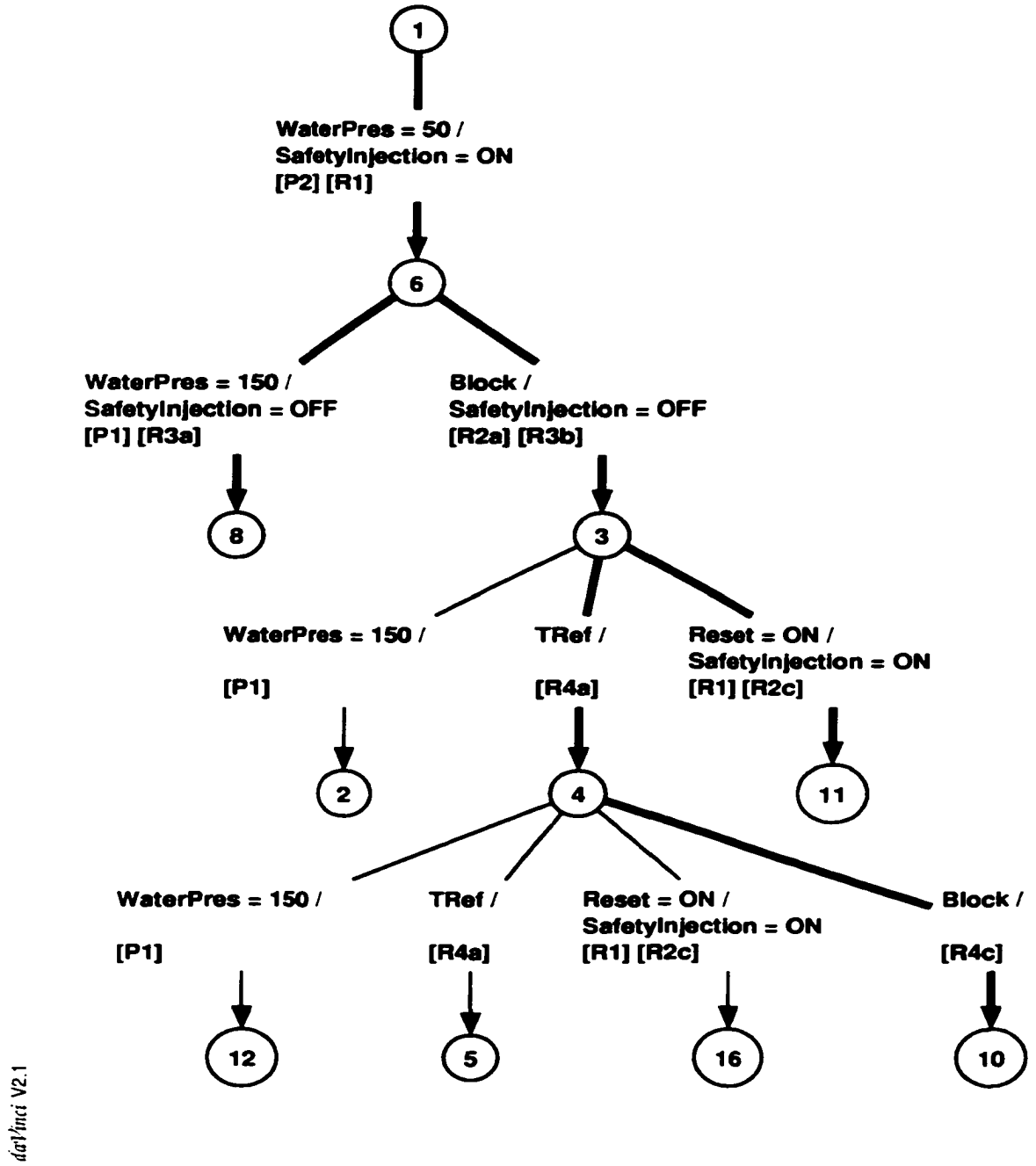


Figure 15. Greedy Scenario Search Tree for Safety Injection System

X is a don't care. At this point the potential states are 2, 5, 12, and 16 (8, 10, and 11 being equivalent to previously expanded states). Counting one for each difference in

enumerated types, the distances for these states are 0, 1, 0, and 2 respectively. There are two states with minimum distance to choose from. With the goal of keeping the lengths of the test scenarios as short as possible, a practical aspect in the implementation of the part 1 algorithm is that it will select the potential state with minimum depth in the scenario search tree in the case of a tie (random select if all tied states are at the same level). As a result the state selected for expansion is state 2. This results in the addition of state 9 and the edge from state 2 to state 9 covers requirement R2b. The back trace from state 9 adds states 9 and 2 to the *ST*.

Targeting the final requirement, R4d, the state required to enable the associated transition is TL, X, 2, X. At this stage the only expandable states are 2, 5, 12, and 16 (8, 9, 10, and 11 being equivalent to previously expanded states). Counting 1 for the enumerated *Pressure* and 2 for the difference in the numeric type *TrefCnt*, the distance for state 2 is 3. Similarly, the distances for states 5, 12, and 16 are 0, 2, and 1 respectively. The distance-based search will expand state 5, adding states 7, 13, 14, and 15 to the scenario search tree. The transition to state 7 is discovered to have covered the target requirement, R4d. The back trace then adds states 7 and 5 to the scenario tree. Noting that state 15 is equivalent to state 1, the values for these addition states are given in Table 11 below.

Table 11. Additional Unique System States at Completion of Part 1 Distance-based

Name	Pressure	Overridden	TrefCnt	SafetyInj
7	TOOLOW	FALSE	3	ON
13	PERMITTED	TRUE	2	OFF
14	TOOLOW	FALSE	2	ON

The base scenario search tree at the completion of the part 1 algorithm is given in Figure 16. Again, the scenario tree, *ST*, is represented by the bold edges.

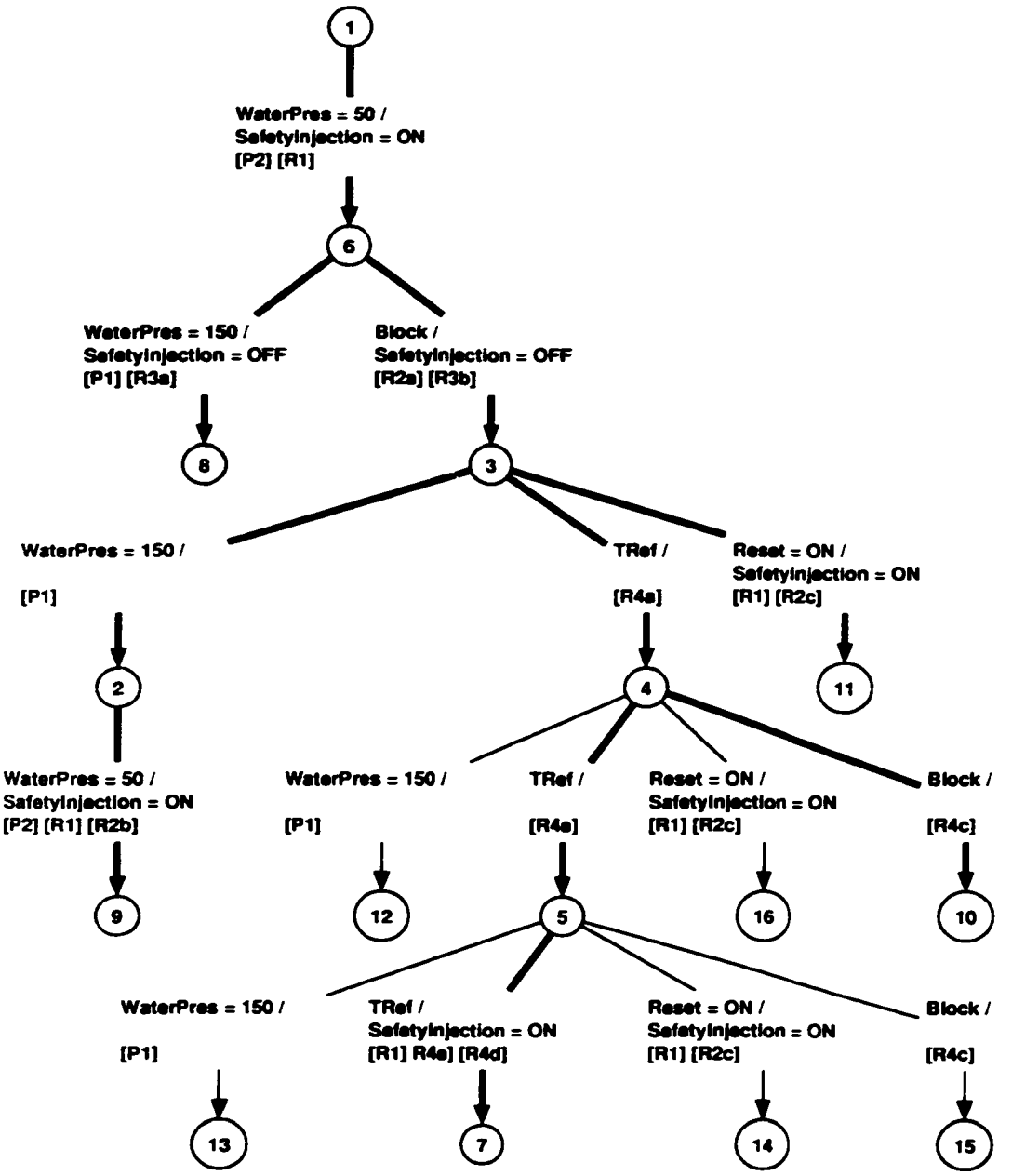


Figure 16. Final Base Scenario Search Tree for Safety Injection System

The generated base scenario tree contains the five scenarios listed below. The complete output from the part 1 algorithm can be found in Appendix B. Note that the implementation of the part 1 algorithm performs a renumbering of the state names so that the names of all states in the final *ST* are number sequentially.

```
GENERATED SCENARIOS: [FORMAT: STATE (EQUIV STATE) INPUT / OUTPUTS]

SCENARIO 1
----------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2] [R1]
     6 WaterPres = 150 / SafetyInjection = OFF [P1] [R3a]
     8 (1)

SCENARIO 2
----------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2] [R1]
     6 Block / SafetyInjection = OFF [R2a] [R3b]
     3 WaterPres = 150 / [P1]
     2 WaterPres = 50 / SafetyInjection = ON [P2] [R1] [R2b]
     9 (6)

SCENARIO 3
----------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2] [R1]
     6 Block / SafetyInjection = OFF [R2a] [R3b]
     3 TRef / [R4a]
     4 TRef / [R4a]
     5 TRef / SafetyInjection = ON [R1] [R4a] [R4d]
     7

SCENARIO 4
----------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2] [R1]
     6 Block / SafetyInjection = OFF [R2a] [R3b]
     3 TRef / [R4a]
     4 Block / [R4c]
    10 (3)

SCENARIO 5
----------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2] [R1]
     6 Block / SafetyInjection = OFF [R2a] [R3b]
     3 Reset = ON / SafetyInjection = ON [R1] [R2c]
    11 (6)
```

4.2 Results of the Part 2 Algorithm

The set of test scenarios in the preceding section cover all requirements identified for the Safety Injection System model. But do these scenarios allow for the verification of all requirements at the black box level? Application of the part 2 algorithm identifies one requirement as not being verifiable for black box testing. That requirement is R4c, which is exercise in the last step of scenario 4. R4c resets the value of *TRefCnt* to zero when *Pressure* is TOOLOW and *Reset* is not On. The previous step in scenario 4 caused *TRefCnt* to be incremented to one, so applying *Block* in the next step should indeed cause a change to the state of the system. This state change cannot be verified by observing system outputs while applying system inputs per scenario 4 as it stands. This is verified by the part 2 algorithm which executes the requirements model with and without the transition associated with R4c disabled and finds no differences at the system outputs. A small portion of the output from the part 2 algorithm is shown below. The complete output can be found in Appendix C.

```
Requirements that have not been verified by the given scenarios:
REQ. ID   Earliest occurrence in SCENARIO ID at LEVEL
-------                          -----------    -----
R4c                                        4        4
```

As a practical consideration, the part 2 algorithm will list the earliest occurrence for each unverified requirement detected. In this case, earliest means that for all scenarios covering the given requirement, the occurrence the fewest number of steps from the start of a scenario is selected. This is done because the list produced by the part 2 algorithm

feeds into the part 3 algorithm as starting points for scenario enhancements. Selecting the earliest occurrence is intended to minimize the length of the final scenarios.

4.3 Results of the Part 3 Algorithm

The part 3 algorithm will target requirement R4c in scenario 4. The requirements model is simulated by applying scenario 4 up to the step that exercised R4c. Then, the input that exercised R4c, *Block*, is applied to the "Good" and "Bad" models (i.e., with the transition associated with R4c enabled and disabled). This provides the initial state pair for the difference-based search.

Figure 17 illustrates the process of applying the difference-based search to produce a verifying sequence for R4c. The states identified by a number indicate the order in which state pairs where (or would have been) expanded. An 'X' following a state indicates that that state pair was not added to the *Potential List*. If the 'X' is covering a number, the new state pair was not added to the *Potential List* because it was equivalent to the pair indicated by that number. If the 'X' is not covering a number, the state pair was not added due to both states in that pair being equivalent (i.e., the difference vanished). The bold arcs indicate the verifying sequence for R4c. The resulting set of states and the enhancement to scenario 4 is shown below.

```
Name  Pressure Overridden TRefCnt SafetyInj
-----------------------------------------
   1    TOOLOW      TRUE        0     OFF
   3    TOOLOW      TRUE        1     OFF
   7    TOOLOW      TRUE        2     OFF

Enhancement for scenario 4:
---------------------------
    1 TRef / [R4a]
    3 TRef / [R4a]
    7
```

```
System output differences detected:
-----------------------------------
SafetyInjection:  Good Value = OFF  Bad Value = ON
```

The difference detected at the system output is also shown. If the value of *TRefCnt* had not been reset to zero as it should have been, *SafetyInjection* would be turned back on one *TRef* event too soon, which is indicated by the "Good" and "Bad" values listed above. The complete output from the part 3 algorithm can be found in Appendix D.

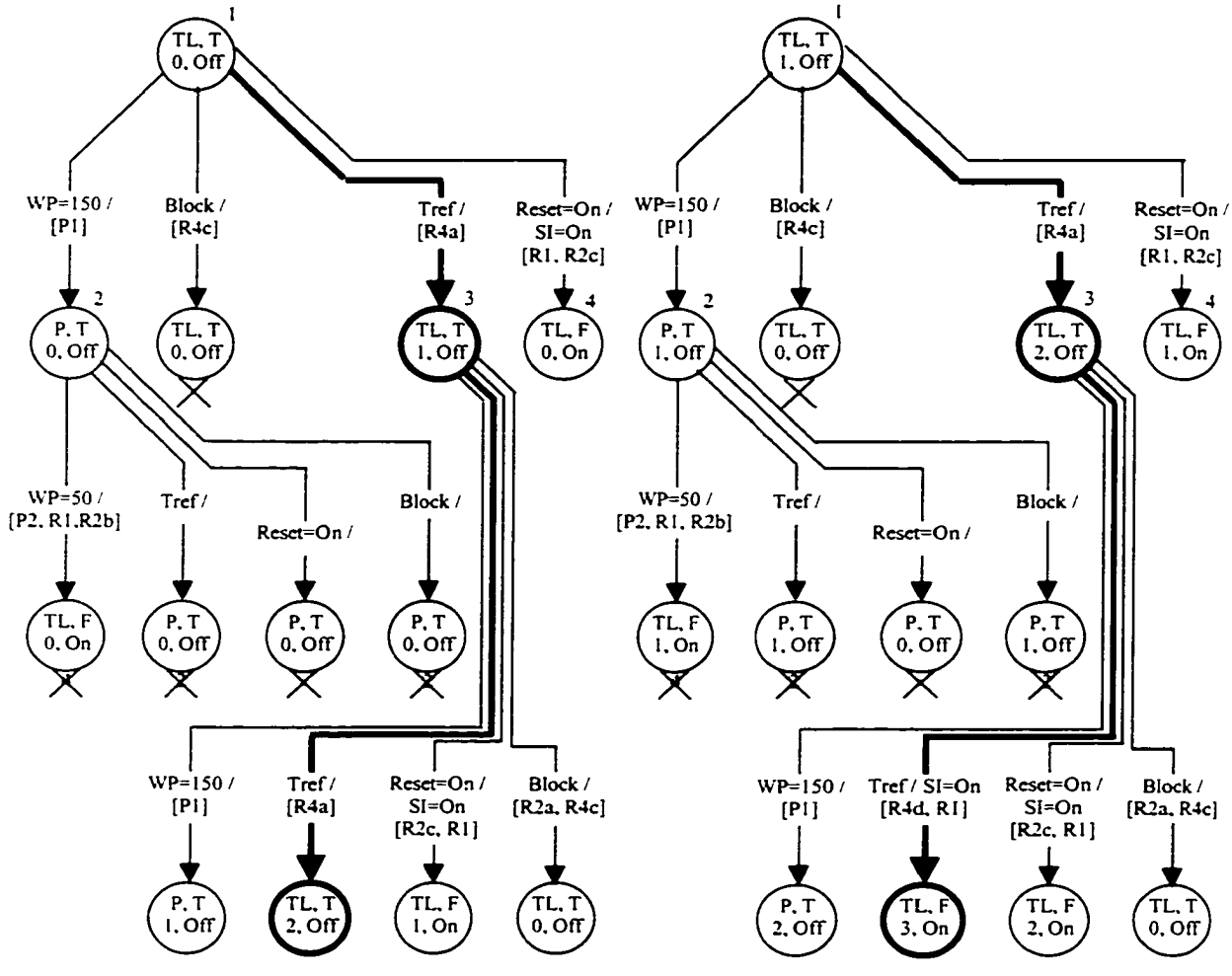


Figure 17. Application of Difference-based Search

4.4 Results of the Part 4 Algorithm

Once the base scenarios and the scenario enhancements have been generated, the part 4 algorithm will combine these results to produce the final set of test scenarios. As described in paragraph 3.4, this is accomplished by adding the scenario enhancements to the base scenario tree to produce the final scenario tree. The final scenario tree for the Safety Injection System is shown in Figure 18. Note that due to the manner in which this tree is constructed (i.e., it is not the direct result of a heuristic search) it will not include potential states. In other words, it is a scenario tree where all states and edges define scenarios, rather than a scenario search tree which includes the results of unproductive search.

The final scenario tree contains the final test scenarios as shown below. The complete output of the part 4 algorithm is given in Appendix E.

```
GENERATED SCENARIOS: [FORMAT: STATE INPUT / OUTPUTS]

SCENARIO 1
----------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2] [R1]
     6 WaterPres = 150 / SafetyInjection = OFF [P1] [R3a]
     8

SCENARIO 2
----------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2] [R1]
     6 Block / SafetyInjection = OFF [R2a] [R3b]
     3 WaterPres = 150 / [P1]
     2 WaterPres = 50 / SafetyInjection = ON [P2] [R1] [R2b]
     9
```

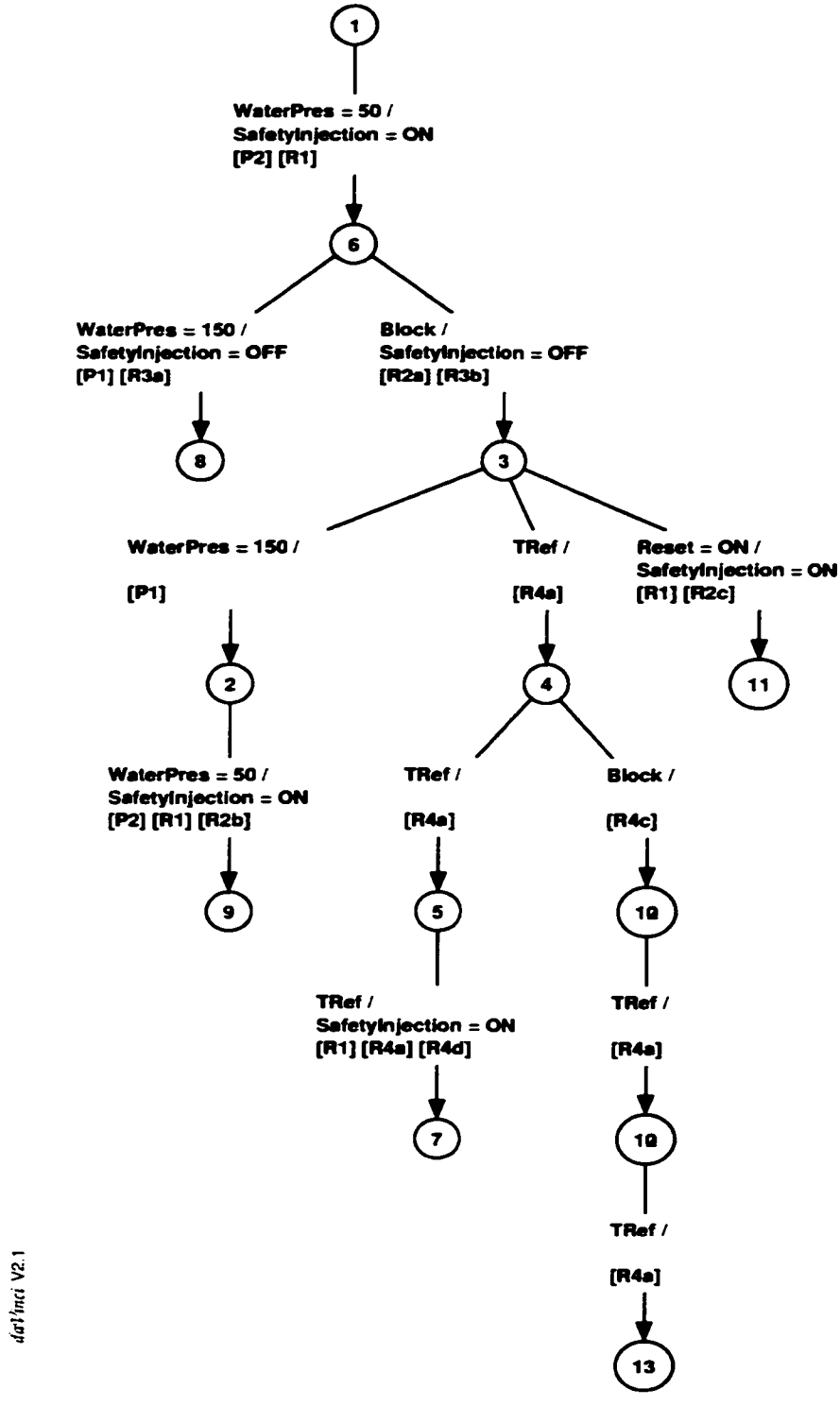


Figure 18. Final Scenario Tree for Safety Injection System

```
SCENARIO 3
----------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2] [R1]
     6 Block / SafetyInjection = OFF [R2a] [R3b]
     3 TRef / [R4a]
     4 TRef / [R4a]
     5 TRef / SafetyInjection = ON [R1] [R4a] [R4d]
     7

SCENARIO 4
----------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2] [R1]
     6 Block / SafetyInjection = OFF [R2a] [R3b]
     3 TRef / [R4a]
     4 Block / [R4c]
    10 TRef / [R4a]
    12 TRef / [R4a]
    13

SCENARIO 5
----------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2] [R1]
     6 Block / SafetyInjection = OFF [R2a] [R3b]
     3 Reset = ON / SafetyInjection = ON [R1] [R2c]
    11
```

5 TREATMENT OF TEMPORAL REQUIREMENTS

Temporal requirements can also be referred to as performance requirements. In general, performance requirements may include requirements on aggregate measures such as minimum throughput or average latency. These types of performance requirements more aptly apply to general computing systems without hard real-time dead lines. For our purposes temporal requirements are requirements that define the hard real-time requirements for the system and possibly for the environment in which the system is intended to operate, if applicable.

In chapter 3 a set of algorithms were presented that will generate a suite of test scenarios from a properly constructed requirements model. These algorithms have been implemented in a set of prototype tools, which are described in chapter 6. The test scenarios define ordered sequences of stimuli and expected responses but do not capture specific temporal relationships between causal events. Testing that a system meets its temporal requirements is a matter of prime importance for real-time embedded systems.

The treatment of temporal requirements has been purposely separated from the scenario generation activity. As previously discussed, function plus performance defines behavior. The generated test scenarios will test proper function and the temporal requirements define proper performance. When the scenarios are modified to include temporal requirements for the environment (i.e., the system under test is stimulated in adherence to the temporal interface requirements), and the results of applying these scenarios are verified against the temporal requirements for the system, the proper behavior of the system can be validated.

This chapter will discuss an approach to the treatment of temporal requirements. The approach involves several different steps in the preparation, application, and analysis of the results of applying test scenarios. Algorithms have been defined for each of these steps. These algorithms have not been implemented, which is discussed in more detail in chapter 6.

5.1 Temporal Requirements for Real-time Embedded Systems

The first order of business here is to precisely capture the temporal requirements. This information will be used in three steps of the process defined in Figure 2 (page 24). The first is in the synthesis of IEEE standard C/ATLAS test programs [63, 64]. This step takes the generated scenarios and the temporal requirements as inputs and generates timed C/ATLAS test programs. C/ATLAS is a general and extensible test language that allows for the unambiguous specification of the test scenarios, including temporal aspects. The C/ATLAS test programs need temporal requirements on the system in order to determine how long to wait for system responses. The second step is the application of the C/ATLAS test programs, which requires the temporal requirement for the environment in order to accurately stimulate the system. The third step is the analysis of test results. This step uses the system temporal requirements to determine if the system has passed the temporal aspects of the tests.

The goal is to define (or adopt) a syntax and semantics that will allow temporal requirements to be described in an unambiguous manner. These descriptions should be easily understood by the engineers designing the system and formal enough to allow for

interpretation by software tools (e.g., the experimental frame that needs to stimulate the system while adhering to temporal requirements placed on the environment).

It will be assumed that all temporal requirements will be defined over a set of events. This assumption is justified by the target systems - embedded real-time systems. These systems are digital and if required to interface to continuous inputs or outputs, discrete sampling or updates will be the method to implement these interfaces.

What are the types of temporal requirements that must be captured? Let's assume for now that all necessary temporal requirements can be defined as a requirement between two unique events. Figure 19 illustrates this situation on a time line.

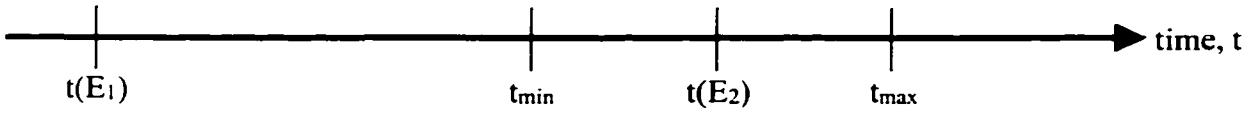


Figure 19. Timing relationships between two events

The events E_1 and E_2 can be either a system input or a system output and are identified by unique names and subscripts. The subscripts define a temporal ordering between events of the same type where E_i occurs prior to E_{i+1}. The three possible temporal requirement types are (the open interval variants are also possible):

1. $t(E_2) \geq t(E_1) + t_{min}$ — E_2 must occur at lest t_{min} after E_1
2. $t(E_2) \leq t(E_1) + t_{max}$ — E_2 must occur not more than t_{max} after E_1
3. $t(E_1) + t_{min} \leq t(E_2) \leq t(E_1) + t_{max}$ — E_2 must occur between t_{min} and t_{max} after E_1.

It seems appropriate to disallow type 1. The reason is that this type gives an open-ended amount of time that could elapse before the occurrence of event E_2. Taken to the extreme, this implies that E_2 need not occur at all (i.e., this is an unbounded time interval). The situations where type 1 would be applied are for events that must wait at least t_{min} and then must occur sometime after that. Instead of using type 1, it should be required that type 3 be used instead. Selection of a t_{max} for these situations may be somewhat arbitrary since there is no real upper bound requirement save that E_2 must occur, probably in some reasonable amount of time. The designer would then specify a "reasonable" t_{max} with the knowledge that it is not a hard requirement and could be changed later if needed. Not using type 1 will simplify the scenario generation problem as well.

Continuing, case 2 may also be eliminated. Allowing t_{min} to be set to zero, or probably more realistically to Δt, where Δt is some minimum allowable time quantum, would allow case 2 to be specified by the case 3 construct. This is because it is assumed that $t(E_2) > t(E_1)$ is always true.

Since all temporal requirements are now defined in terms of a bounded interval, the requirements could be specified as such (with variants of open and closed end points):

3. $t(E_2) \in [t(E_1) + t_{min} , t(E_1) + t_{max}]$

5.2 Specification of Temporal Requirements

Capturing the temporal requirements for a system should be a straight forward process. Capturing these requirements will also help identify missing requirements since

the temporal requirements applying to the system must be completely specified for the proposed test generation method to work. This should not be considered a major hurtle. If certain temporal relationships are not bound to hard real-time deadlines, "reasonable" tolerances may be specified to support the test generation process. Suitable requirement traceability should identify these types of temporal requirements. Note that the temporal requirements applying to the environment can be left incompletely specified, in which case the test environment will be free to apply the stimulus as quickly as possible after the enabling event.

The process of specifying temporal requirements is best illustrated by example. The Safety Injection System (SIS) will once again be used. The first step is to analyze the stated functional requirements, which were used to generate the test scenarios. Using the SCR representation for events (@C(A) specified the time at which A changes value), an example set of Stimulus-Response temporal requirements for the SIS are given in Table 12 below.

Table 12. Stimulus-Response Temporal Requirements for SIS

E_1	E_2	t_{min} (seconds)	t_{max} (seconds)
@T(*WaterPres* < LOW)	@T(*SafetyInjection* = On)	0	1
@C(*Tref*)	@T(*SafetyInjection* = On)	0	3
@T(*Reset* = On)	@T(*SafetyInjection* = On)	0	2
@C(*Block*)	@T(*SafetyInjection* = Off)	0	1.5
@T(*WaterPres* ≥ LOW)	@T(*SafetyInjection* = Off)	0	0.5

The minimum and maximum temporal constraints given may or may not be reasonable but suffice for illustrative purposes and were chosen to be unique for easy identification in the C/ATLAS test programs.

Each row in Table 12 represents one temporal requirement. For example, using the notation presented in section 5.1, row one represents the following:

$$t(@T(\mathit{WaterPres} < \text{LOW})) \leq t(@T(\mathit{SafetyInjection} = \text{On})) \leq t(@T(\mathit{WaterPres} < \text{LOW})) + 1$$

For the SIS it is appropriate to specify some Stimulus-Stimulus temporal requirements as well. These types of requirements may represent interface requirements in computer to computer communication (e.g., the time between *Tref* events) or may represent an expected sampling interval and the corresponding limits of physical phenomenon (e.g., the time between changes in *WaterPres*). Where humans are involved, these types of constraints are not as meaningful except for the minimum time, which may represent the fastest possible consecutive button presses. The maximum times, if not based on an interface requirement, are specified as reasonable times, keeping in mind that test time will be effected by the values chosen. The SIS does not have any Response-Stimulus or Response-Response temporal requirements.

Table 13. Stimulus-Stimulus Temporal Requirement for SIS

E_1	E_2	t_{min} (seconds)	t_{max} (seconds)
@C(*WaterPres*)	@C(*WaterPres*)	1	5
@C(*Tref*)	@C(*Tref*)	10	10
@C(*Block*)	@C(*Reset*)	0.5	5
@C(*Reset*)	@C(*Block*)	0.5	5

5.3 System Input Timings Within Test Scenarios

Having the temporal requirements specified, they must now be used to add temporal constraints to the test scenarios previously generated. Temporal requirements either apply to the system or to the environment. In relation to test, the requirements placed on

the environment determine when the experimental frame (EF) or real-time test environment (RTTE), which is emulating the environment of the system, will apply stimulus to the system under test. Temporal requirements placed on the system are used by the EF or RTTE to determine how long to wait for system responses and during data analysis to determine if the system has passed a given test. This section is concerned with the environmental requirements because we are defining how the EF or the RTTE will determine when to apply stimulus to the system.

The initial plan was to specify temporal scenarios such that the EF or RTTE interpreting them would be able to provide system inputs (from the generator) without dependencies on system outputs. This would ensure that a scenario could be successfully applied to a system even in the presence of system errors (e.g., outputs occurring at the improper time or not at all). To achieve this, the inputs specified in the scenario would have to be tagged with an absolute time, probably relative to the start of the test, at which they should be applied. But can these time tags be generated independently of system behavior (i.e., statically prior to test execution)?

There is certainly latitude here since the temporal requirements as formulated above define an interval in which the events may occur. Dasarathy [65] has characterized temporal requirements as relations between stimulus and response. Here we are assuming a system-centric point of view where system inputs are stimulus and system outputs are responses. The events, E_1 and E_2, may be either stimulus or response giving four possible combinations. Since we are discussing the timing of stimulus in this section, the stimulus-stimulus and response-stimulus are the only two cases of interest.

These cases define requirements imposed on the environment and will be used by the generator of the EF or RTTE. The other two cases, response-response and stimulus-response, define requirements imposed on the system and these are to be verified by the experimental frame or during post processing of the collected experimental data.

The stimulus-stimulus case is the easiest to deal with since both events originate in the environment. From a testing perspective, this is the part over which we have control. The application times for E_2 would be:

$$t(E_2) = t(E_1) + t_{min} \qquad \text{or} \qquad t(E_2) = t(E_1) + t_{max}$$

These represent the minimum and maximum timings. The justification for the selection of these application times will be discussed later in this section.

The response-stimulus case presents a problem. In this case the application time for event E_2 is based on the response of the system. Even though the time at which each such response occurs is bounded for correct operation of the system, it is not possible to determine $t(E_2)$ without prior knowledge of the actual time of the system response for E_1. The reason is that the interval defined by $t_{max} - t_{min}$ associated with E_1 may be greater than t_{max} associated with E_2. The interval $t_{max} - t_{min}$ represents the range of possible values of $t(E_x)$ for a system exhibiting correct behavior. If the generator assumes E_1 occurs at t_{min}, it would be possible that, for some correctly operating systems, E_2 would occur prior to E_1. If the generator assumes E_1 occurs at t_{max} then it is not possible to guarantee that E_2 would be applied within the temporal requirement for E_2. Even if this argument could

not be made, the error associated with the uncertainty for $t(E_1)$ would effect all subsequent events in the scenario and "stack up" as the test scenario progressed.

The problematic situation is illustrated in Figure 20 below. Suppose that the temporal requirement for the response E_1 is that it must occur between 3 and 20 time units after E_0. Also suppose that the temporal requirement for stimulus E_2 is that it must occur between 3.99 and 4.01 units after E_1. Assuming E_0 occurs at time 0, E_1 must occur between 3 and 20 to meet requirements. If the generator assumes the minimum time for E_1, then E_2 would be applied at about 7 (for simplicity only one E_2 is shown in the figure). The system under test may certainly respond with E_1 anytime between 7 and 20, which still meets requirements, so applying E_1 in this manner may cause E_2 to occur before E_1! If on the other hand, the generator assumes the maximum time for E_1, then E_2 would be applied at 24. In this case if the system responds with E_1 any time prior to 20, the test environment will fail to meet the environmental temporal requirement for the relation between E_2 and E_1. Assuming any other valid time for E_1 results in various combinations of these two problems.

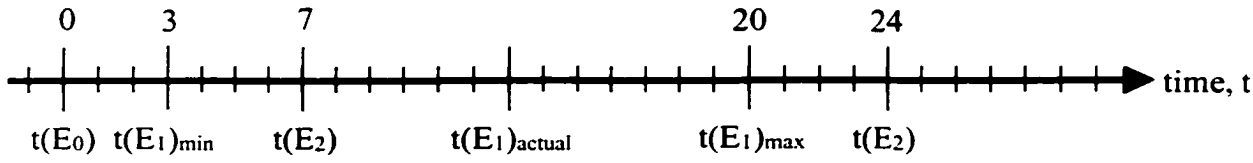


Figure 20. Example of problematic temporal relationships

The result here is that the original approach is not feasible. What this implies is that the generator in the experimental frame must be adaptive and respond to the timing of system outputs when necessary. Instead of tagging the stimulus with absolute time values, the stimulus should be tagged with the appropriate temporal requirement. The generator will then be required to compute the application times for stimulus during execution.

So how will this adaptive generator work? There are four possible cases for the E_1 response of the system. E_1 may be generated early (within $(t(E_0)\ ,\ t(E_0) + t_{min}]$), on time (within $[t(E_0) + t_{min}\ ,\ t(E_0) + t_{max}]$), late (within $(t(E_0) + t_{max}\ ,\ \infty)$), or not at all. It will be assumed that it is desirable to allow the test scenario to always execute to completion. The test could also be halted on user defined criteria such as on the first failure, after ten failures, etc. The ability to halt based on test results assumes that the test environment is capable of evaluating test results on the fly at runtime. There are certainly pros and cons to this approach. If the test runs to completion, this allows as much data as possible to be collected. Even though the validity of tests performed after a failure may be in question (just think of compiler errors after the first one or two syntax errors), the data may still be insightful in some cases. The magnitudes of the temporal requirements and the lengths of the test scenarios are also very important. If the temporal requirements are on the order of seconds with test scenarios lasting minutes or hours, it may be possible for a fairly inexpensive test environment to check test results on the fly, possibly saving valuable test time. On the other hand, if the temporal requirements are generally on the order of micro-seconds with test scenarios lasting seconds, it may take a rather expensive test

environment to be able to perform on the fly evaluation. In this situation, the potential for savings of test time may not warrant the expense of the test environment that would allow for on the fly test evaluation.

With this in mind, the generator response to E_1 events that occur early or on time should be per the requirement for E_2. For the late case, it might be possible and desirable to wait and allow proper timing for E_2 as long as E_1 is not too late. What is too late will be constrained by what is deemed to be a reasonable test time for a given scenario. This could be specified as a parameter, either globally to be applied to all requirements (e.g., $2*t_{max}$ or $t_{max} + 4.5$), or could be defined on a per requirement basis. Once the specified maximum time has elapsed since E_1, the generator would apply E_2 assuming that E_1 had occurred at this maximum time. This will also cover the case where E_1 is not generated at all.

With the knowledge that these scenarios will be used as test cases to validate system designs, the test criteria should be selected such that the most stressful cases are applied to the system. What is most stressful is also an open issue. Usually one would expect that the minimum timing would be most stressful. However, this may not be the case. For example, if a system is forced to wait until t_{max} for a particular input, an internal timer overflow might cause the system to fail to properly respond to the input.

This argument suggests that perhaps both minimal and maximal timing should be applied to the system. If this is attempted the result will be a 2^n explosion in the number of test scenarios, where n is the number of inputs to be applied, in order to cover all possible min-max combinations. An alternative would be to apply two scenarios, one

using all minimal timings and one using all maximal. This would of course only cover two of the combinations, but should provide a fairly robust test set. Another approach might be to allow the designer (or test engineer) to specify, on a requirement by requirement basis, whether the minimum or maximum timing is the most stressful case.

5.3.1 Time in SCR Model Specifications

There is one additional issue that must be addressed before the C/ATLAS test programs can be synthesized. This issue is how to deal with time in an SCR model. Strictly speaking, SCR models do not include time. The exception is the *duration*() operator [49] which allows the modeler to specify how long a particular condition must hold before some action is to be taken. Unfortunately this construct is not general enough to allow all types of timed model behavior to be specified.

Having access to time can allow a real-time system to take required actions at specified times from the enabling event(s). Examples of these types actions are error recovery taken a certain amount of time after some failure or an output that is to be sent or updated at a periodic rate. Typically the system will contain a time reference source, usually an oscillator with a known rate, that can be used to monitor the advancement of time. This type of construct is not allowed in SCR model specification because it violates circular dependency rules enforced by the SCR tool set's model checker.

The method that has been used to work around this issue is that of abstracting the time reference mechanism out of the model. This approach has been implemented in two flavors. The first is illustrated in the example Safety Injection System where time was needed to allow the system to automatically unblock itself. In this case it was assumed

that the time reference mechanism was physically outside the system and supplied timing reference events to the system at a specified rate. The system then only has to count the events to track the progress of time. This approach is straight forward since the timing input is treated exactly like any other systems input. The possible disadvantage is that this approach requires that the formal interface specification actually contain this signal.

The other approach is to designate *Time*, when used as a monitored variable name, to be a reserved word and give it unique functionality. This input represents an artificial input to the system (i.e., not part of the formal interface specification). The special functionality will be provided during the process of synthesizing code from the SCR model, which is described in chapter 6.

The special treatment of time is needed due to differing needs of the requirements model and the process of synthesizing the C/ATLAS test programs. The SCR model needs to have access to absolute time, usually referenced to system start. The C/ATLAS synthesis process needs a delta time from the most recent time reference in order to execute a wait of the proper amount of time. Absolute times cannot be used in the C/ATLAS test programs because the application of the test scenarios by the test environment is necessarily adaptive as described in section 5.3.

The special treatment of time is handled by the code synthesis process. When a model is processed that contains a monitored variable named *Time*, the synthesized code will treat the test values for time as deltas to the current time for each stored model state. The model will be supplied with the resulting absolute time value, but the reported test scenarios will show the relative time deltas that where applied.

This approach was used in the elevator controller model, which will be described in a later chapter. Because *Time* is not part of the actual interface to the system, models or prototypes of the real system that are being tested will contain their own time references. As a result, during the synthesis of the C/ATLAS test programs, the *Time* assignments that exist in the test scenarios will be converted to WAIT statements which will cause the test environment to wait the indicated amount of time.

5.3.2 Synthesis of C/ATLAS Test Programs

C/ATLAS is a generic test specification language originally captured in IEEE STD 416-1978. This language originated in the avionics field with the acronym standing for Abbreviated Test Language for Avionics Systems. This language has been maintained as C/ATLAS and is now under IEEE STD 716-1995 [63, 64]. The acronym now stands for Common/Abbreviated Test Language for All Systems.

When the test scenarios have been generated and the temporal requirements have been captured, the C/ATLAS test programs may be synthesized. The synthesis process will take the test scenarios and the SCR specification, which includes the temporal requirements, as input. Figure 21 provides a top level algorithmic description for this synthesis process.

The final step is intended to allow the test environment to record any system response that might occur after the last input has been applied. However, since in general a previously applied input might cause a system response that is required to occur after the response enabled by the final input, a more robust approach would be to set the time

of the final WAIT FOR statement to the largest t_{max} for all Stimulus-Response and Response-Response temporal requirements for the system.

```
for all scenarios
{
  Open a new output file.
  Output BEGIN statement and header information from SCR specification.
  Output model independent C/ATLAS extensions (abstract signals & dataless
     events).
  Output SIGNAL definitions by converting data in SCR variable dictionary
     for monitored (input) and controlled (output) types.
  Output EVENT identifiers for all expected system responses by scanning the
     present test scenario.
  Output APPLY statements for initial states of monitored variables as
     defined in the SCR variable dictionary.
  Output ENABLE statements for monitoring of all defined events.
  for each step of the current scenario
  {
    if (specified monitored variable is not Time)
      Output APPLY statement to set value of specified monitored variable.
    if (current step specifies an expected change for one or more controlled
        variables)
    {
      for all listed controlled variables
        find the temporal requirement with the largest t_max and E_1 that is
        compatible with the most recent monitored or controlled variable
        event and E_2 that is compatible with one of the present controlled
        variable events and output a WAIT FOR event statement with the event
        set to E_2 and MAX-TIME set to t_max.
    }
  }
}
```

Figure 21. C/ATLAS Synthesis Algorithm Top Level Structure

Appendix F contains manually generated C/ATLAS test programs for the Safety Injection System based on the generated test scenarios presented in section 4.4 and the temporal requirements given in section 5.2.

5.3.3 Interpretation of C/ATLAS Test Programs

For the reasons described in section 5.3, the C/ATLAS test programs do not contain all of the temporal information needed to allow the test environment to stimulate the

system per the environmental temporal requirements (Stimulus-Stimulus or Response-Stimulus). The additional information is the temporal requirements themselves.

An input parameter, MAX-MIN, will be used to determine whether maximum or minimum stimulus timing will be used when applying the test scenario. Assuming that the test environment is maintaining the current global test time and all system responses are time tagged and logged, a high level representation of the algorithm that should be used by the test environment (e.g., the generator in an experimental frame) to apply the test scenarios is given in Figure 22.

One situation that must be handled is when a system response comes late. The generator may have used the specified maximum allowable time for $t(E_1)$ when determining when to apply E_2. After this decision, E_1 may actually arrive. If another stimulus, say E_3, is also to be applied at a time relative to E_1, in order to remain consistent, the assumed time for E_1 used when determining $t(E_2)$ should also be used to determine $t(E_3)$. The implication is that the generator should be able to keep track of the fact that a system response time is assumed or actual. If an actual time is supplied to the generator when an assumed time exists for the same system response, the actual time should be ignored.

Processing temporal requirements at test time may present an unacceptable overhead (i.e., the search for the appropriate requirement). This situation could be improved by performing a more thorough analysis during C/ATLAS test program synthesis. These test programs could be annotated with statement numbers to identify the enabling event

```
Apply the initial values to all monitored variables (as specified in
   "Set up initial input values" section).
Enable monitoring of all identified events (as specified in "Enable
   monitoring of events" section).

for each C/ATLAS statement (as specified in "Start the test scenario"
         section)
{
  if (the statement is WAIT FOR time)
    Delay for the specified time.
  if (the statement is WAIT FOR event)
  {
    Delay until the event is detected or until MAX-TIME has elapsed.
    if (event was detected)
      Add the event and the time at which it was detected to the test
      output log.
  }
  if (the statement is an APPLY statement)
  {
    Scan the present test history backward from the present test time
    for all compatible S-S and R-S temporal requirements (E1 found in
    test history and E2 from current APPLY statement).
    if (at lease one compatible temporal requirement was found)
      delay for either the largest tmin or tmax (as specified by MIN-
      MAX) of all compatible temporal requirements.
    Set the value of the appropriate monitored variable as specified
    and add this event the time of application to the test output
    log.
  }
}

Scan the present test history backward from the present test time for
all S-R and R-R temporal requirements with E1 compatible with an
event in the test history (E2 is a don't care) and determine the
largest value for tend = t(E1) + tmax.
if (tend is greater than the present test time)
  Delay for tend - present test time.
```

Figure 22. TE Scenario Application Algorithm Top Level Structure

and required timing so that the test environment can quickly calculate the proper time for application of each input event.

5.4 Analysis of Test Results

Now suppose that the C/ATLAS test programs have been executed by a suitable test environment and the test scenarios have been applied to a model or a physical system, usually referred to as the model under test (MUT) or system under test (SUT). In the proposed method, the test environment will have time tagged all events, both stimulus and response, and logged them for post-test analysis.

The analysis of the collected test data is analogous to the application of the test scenarios themselves. The similarity is that during the application process, response events are matched with temporal requirements in order to determine when to apply a stimulus that is constrained with respect to the response. When test data is analyzed, the response events are matched with temporal stimulus requirements in order to determine if the response occurred within the required temporal bounds.

The input to the analysis process will be the test log of collected test data, which is assumed to be sets of time-event pairs, the test scenarios (or C/ATLAS test programs if they prove more convenient), and the temporal requirements for the system. A top level description of the algorithm to perform this analysis is given in Figure 23.

Returning to the Safety Injection System, the use of this algorithm will now be illustrated with an example. Recall test scenario 1 and the stimulus-Response temporal requirements for the Safety Injection System, which are repeated below for ease of reference.

```
for each line of the test log
{
  if (the current line specifies a system response)
  {
    Scan the test scenario and locate the next occurrence of an
    equivalent response event.
    if (the test scenario does not contain an equivalent response)
      Output a test failure message.
    else
    {
      Mark the equivalent responses in both the test scenario and the
          test log.
      Scan the test log backward from the present time to locate the
          most recent event compatible with a temporal requirement E1
          event and with E2 compatible with the present response
          event.
      if (a compatible temporal requirement cannot be found)
        Output an error message regarding the missing requirement.
      if (the logged times of the E1 and E2 events do not meet the
          temporal requirement)
        Output a test failure message.
    }
  }
}

Output a test failure message for all unmarked response events in the
test scenario and the test log.
```

Figure 23. Test Data Analysis Algorithm Top Level Structure

```
SCENARIO 1
----------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2] [R1]
     6 WaterPres = 150 / SafetyInjection = OFF [P1] [R3a]
     8
```

Table 14. Stimulus-Response Temporal Requirements for SIS

E_1	E_2	t_{min} (seconds)	t_{max} (seconds)
@T(*WaterPres* < LOW)	@T(*SafetyInjection* = On)	0	1
@C(*Tref*)	@T(*SafetyInjection* = On)	0	3
@T(*Reset* = On)	@T(*SafetyInjection* = On)	0	2
@C(*Block*)	@T(*SafetyInjection* = Off)	0	1.5
@T(*WaterPres* ≥ LOW)	@T(*SafetyInjection* = Off)	0	0.5

Suppose that a test environment has collected the data shown in Table 15 after applying test scenario 1.

Table 15. Test Data from Safety Injection System Test Scenario 1

Row	Time Stamp	Observed Input and Output Changes
0	t_0 (Initial conditions)	*WaterPres* = 150, *Reset* = Off, *Pressure* = PERMITTED, *SafetyInjection* = Off
1	t_1	*WaterPres* = 50
2	t_2	*SafetyInjection* = On
3	t_3	*WaterPres* = 150
4	t_4	*SafetyInjection* = Off

The algorithm will identify row 2 as containing a system response (row 0 will be skipped as the special case of the initial conditions). The first step of scenario 1 is equivalent to this logged event, resulting in both being marked to indicate the matching between these events. The test log is then scanned backward to determine that the E_2 event for this response is that *SafetyInjection* has changed from Off to On. The temporal requirements in the first three rows of Table 14 are compatible with the current E_2. Now scanning the test log for a compatible E_1, it can be determined that at row 1 *WaterPres* changed from 150 to 50. This is compatible with the E_1 event in the temporal requirement from row 1 of Table 14. Having identified the temporal requirement, the logged times can then be checked. Substituting the known quantities into the previously given formulation of the temporal requirements, with $t_1 = t(E_1)$ and $t_2 = t(E_2)$ the expression to be evaluated is:

$$t_1 + 0 \leq t_2 \leq t_1 + 1$$

If the recorded values for t_1 and t_2 satisfy the expression above, then this test has passed. If not, a test failure message will be output.

Evaluation of the other recorded system response, *SafetyInjection* = Off at row 4 of Table 15, proceeds as described above. The fifth row of Table 12 is identified as the correct temporal requirement and t_3 and t_4 are the times for E_1 and E_2.

If there had been a mismatch in the system responses between the test scenario and the test log, this situation would be identified by the process of marking equivalent responses. The final step of the algorithm would report any responses that were expected but not recorded and any responses that occurred but were not expected. In other words, there must be an exact matching between expected and recorded system responses to the test scenario stimulus.

6 TOOL SUPPORT

In this chapter the topic of software tool support for the test generation process described in the previous chapters will be discussed. Adequate tools to support this process are essential to enable practical use. Tools will also allow the process to be automated to the greatest extent possible.

Each section is devoted to one of the steps of the test generation process illustrated in Figure 2 (this chapter frequently references this figure to identify where each tool fits into the process) suitable for automation through the use of a software tool. Recall that this process is illustrated in Figure 2 (page 24). This happens to be all steps except for the initial step of developing the requirements model from the text-based requirements. The algorithms presented in earlier chapters define the tasks that must be carried out at almost all of the process steps. As a result, at the most fundamental level, the tools developed simply implement these algorithms. Each section describes the state of the prototype tool and discusses practical considerations for developing the needed tool if one does not exist.

6.1 Requirements Model Code Synthesis

Automation of the task of synthesizing executable code from the requirements model is the purpose of the Requirements Model Code Synthesis (RMCS) tool. This task is identified in Figure 2 by the label ①. In addition to the desire to automate the complete test generation process to the greatest extent possible, the reasons for automating this task are numerous. First, the task of synthesizing the needed code is very mechanical and is thus very suitable for automation. Secondly, this task is very tedious and error prone if

performed manually. Finally, because it is necessary for the system being simulated during scenario generation to accurately represent the requirements model, an argument could be made that this task *must* be automated in order to produce correct test scenarios.

Figure 24 provides a graphical representation of the inputs and outputs for the RMCS tool.

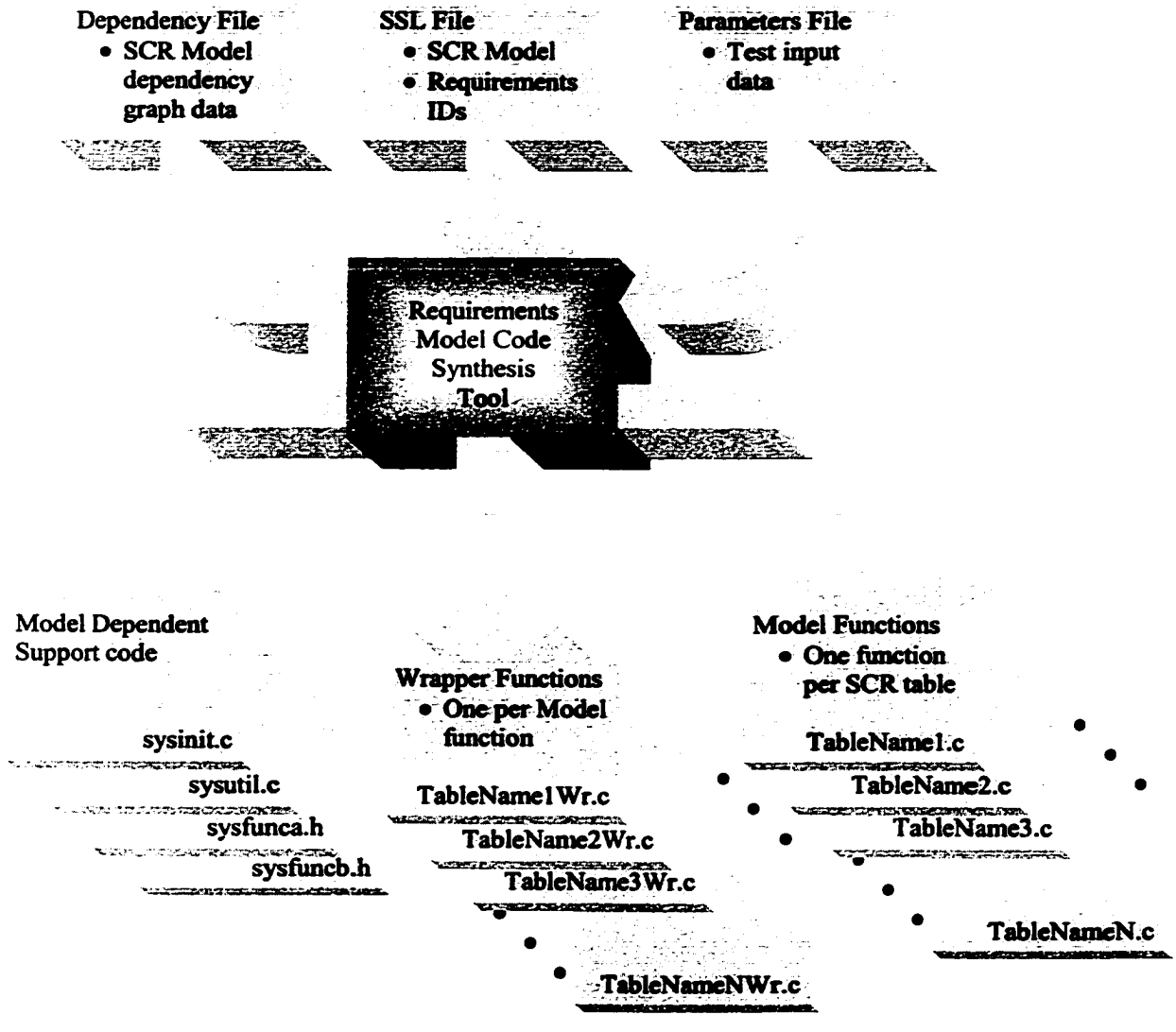


Figure 24. Requirements Model Code Synthesis Tool

6.1.1 Inputs to RMCS tool

There are three sets of inputs. The first is the SCR dependency graph information. This file contains a tabular representation of the information provided graphically by the Dependency Graph Browser within the SCR tool set. Although the version of the SCR tool set used for this research does not provide the capability to automatically generate this file, the second generation of the SCR tool set, which is in beta testing at the time of this writing, will provide this capability.

The dependency graph data file for the Safety Injection System model is listed below. The first eight lines simply enumerate all of the model variables along with their types. The remaining lines provide each individual dependency, which, thinking graphically, would represent each edge in the dependency graph. The dependent variable is listed first followed by the variable on which it depends. For example, *Overridden* depends on *Pressure*, *Block*, *Reset*, and *TRef*.

```
modeclass Pressure
monitored Block
term Overridden
monitored Reset
controlled SafetyInjection
monitored TRef
term TRefCnt
monitored WaterPres
Pressure, WaterPres
Overridden, Pressure
Overridden, Block
Overridden, Reset
Overridden, TRef
SafetyInjection, Pressure
SafetyInjection, Overridden
TRefCnt, Block
TRefCnt, TRef
```

The RMCS tool uses the dependency graph information in this file to synthesize a static dependency matrix. This matrix is used by the simulation engines in the implemented scenario generation algorithms discussed in the next section.

The second input to the RMCS tool is the *SSL* file for the SCR Requirements model. The *SSL*, which is presumed to stand form SCR Specification Language, is an ASCII representation of the SCR model information. The grammar of *SSL* is provided in Table 13 of the draft SCR*Toolset: The User Guide[2]. The draft user guide also provides the grammar of the expressions contained in the *SSL* in Table 9. While the RMCS tool was able to use the *SSL* grammar as is, some problems where encountered with the expression grammar that required this grammar to be altered slightly. Appendix G contains the altered grammar that the RMCS tool will accept.

In addition to the normal SCR model information contained in the *SSL* file, the requirement identifiers are included in the *Description* blocks. The RMCS tool finds these identifiers within the *Description* blocks (in case this block contains other free-form text) by locating the string "Requirements Identifiers:" which can be seen in the *SSL* file included in Appendix A. The requirement identifiers should then follow on the next r rows, where r is the number of rows in the corresponding SCR table. Each row should contain c requirement identifiers separated by white space, where c is the number of columns in the corresponding SCR table.

---

[2] Provided courtesy of the Naval Research Labs

The final input to the RMCS tool is a parameter file. For the present version of the RMCS tool, this file only defines the name of the system and the test values for all monitored variables. This file is intended to include any parameters that might be needed for control of future tool enhancements. The parameter file for the Safety Injection System is listed below. The system name (i.e., full or long name) is used in output header blocks defined in the synthesized code and the short name is used for file naming purposes. This file also shows the two possible methods of specifying the test values to be used during simulation. The first is to explicitly define the values by enumerating them. This style is identified by the *absolute* keyword. The second method is to define delta values that will be applied to the most recent value for that monitored variable as captured in the recorded state during simulation. This method is identified by the *delta* keyword which is followed by the delta values (either positive, negative or both) and a specified min,max range. The style of the parenthesis used for the min,max range determines whether the endpoint is included. In addition, if the monitored variable name is prepended with an '*' then this input will be treated as part of the system state for purposes exploring state-spaces by the implemented scenario generation algorithms.

```
System Name: Safety Injection System
ID: sis
WaterPres delta: +/-100 [50 150]
TRef absolute: Value1 Value2
Reset absolute: Off On
Block absolute: Value1 Value2
```

6.1.2 Outputs of RMCS tool

The output of the RMCS tool is C code. This code is ready to compile, link, and execute with the scenario generation algorithms. This code can be divided into three main groups. The first and most basic part is the set of files making up the SCR table functions. Each table in the SCR model defines a function controlling one model variable. The synthesized code represents the equivalent function expressed in C. In addition, this code includes "instrumentation" that allows the scenario generation algorithms to easily determine when state transitions associated with each requirement identifier has been exercised and to disable these state transitions when needed by the part 2 and part 3 algorithms.

The second main group of code is a set of "wrapper" functions. These functions provide a generic interface between the scenario generation algorithms and the actual model functions. This is needed so that the scenario generation algorithm code can be model independent. The wrapper functions also perform some bookkeeping such as checking the instrumentation and logging any state changes after execution of the model function.

The final main group of synthesized code is a variety of model dependent support functions and two header files. The header files, *sysfunca.h* and *sysfuncb.h* define the interfaces between the scenario generation algorithms and the synthesized code and between the support code and the function code respectively. The model dependent support code includes functions to initialize state structures, set the list of available input values, compare states, copy states, compute the distance to a the enabling state for a

target requirement (used by the distance-based search algorithm), and various input and output support functions.

An additional output from the RMCS tool not shown in Figure 24 is a set of make files [66]. These make files are used to manage the compilation of the synthesized model code and the scenario generation algorithms to produce executables for the four parts of the scenario generation process, which is described in more detail in section 6.2 below.

As a point of reference, the combined number of lines in the synthesized code from the RMCS tool for the safety injection system model is approximately 1800.

6.1.3 Prototype RMCS tool

A prototype of the RMCS tool has been implemented. This implementation is based on the compiler development tools lex and yacc [67] and the supporting code has been written in the C programming language [68]. This prototype tool consists of about 5500 lines of C and processes the inputs and generates the outputs as described in the previous two sections.

6.2 Scenario Generation Algorithms

Automation of the generation of test scenarios is the task to be performed by the scenario generation algorithms. This task is identified in Figure 2 by the label ②. A set of prototype implementations of the scenario generation algorithms, as defined in chapter 3, have been developed in the C programming language. The C code synthesized by the RMCS tool has been designed to work with these algorithm implementations. Figure 25 illustrates how these algorithm implementations are combined with the model and

support code and how the algorithms interact. As described in chapter 3, the complete scenario generation process involves executing the four algorithms sequentially. These four algorithms have been referred to as the scenario generation Part 1, Part 2, Part 3, and Part 4 algorithms. These four algorithms are implemented by *sga.c* (Scenario Generation Algorithm), *sva.c* (Scenario Verification Algorithm), *sea.c* (Scenario Enhancing Algorithm), and *sca.c* (Scenario Combining Algorithm), respectively. Figure 25 shows how the result from each algorithm is used to support subsequent algorithms.

As Figure 25 shows, the synthesized code from the RMCS tool is consolidated into a library called sysfunc.a. This library is then compiled and linked with a graph package library, a list package library, an input/output support library, and each individual algorithm to generate model unique executables that, when executed, will apply the each algorithm to the given model. Once compiled these executables are self contained (i.e., they do not require any further inputs).

The C code needed for the prototype implementations of the base scenario generation algorithm, the requirements verification algorithm, the scenario enhancing algorithm, the scenario combining algorithm, the graph package, the list package and the I/O library are approximately 900, 700, 1300, 800, 1500, 400, and 1900 lines respectively.

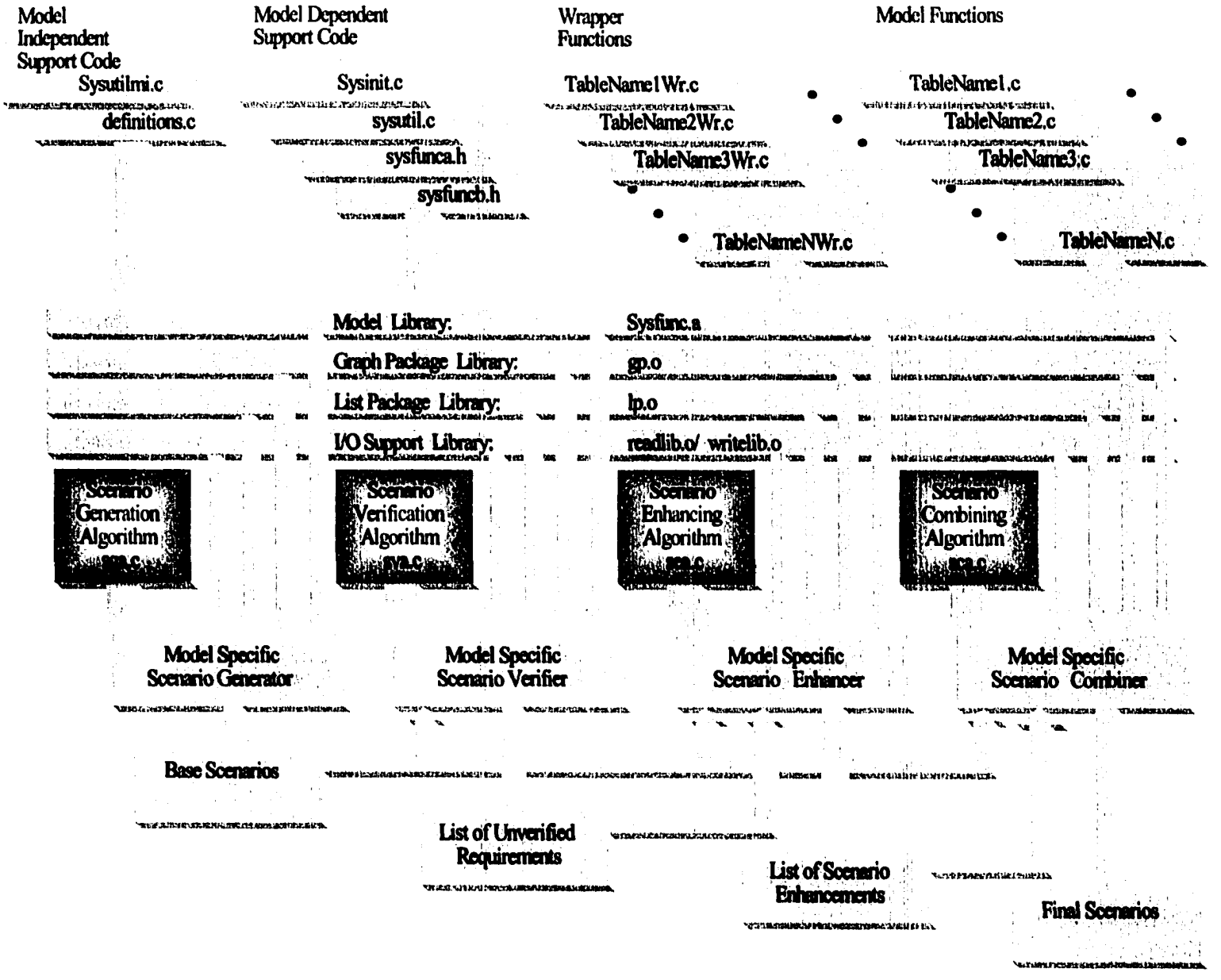


Figure 25. Scenario Generation Algorithms

6.3 C/ATLAS Test Program Synthesis

Automation of the task of combining the generated test scenarios with the temporal requirements and producing a test description in a standard test language is the purpose of the C/ATLAS Test Program Synthesis tool. This task is identified in Figure 2 by the label ③. A C/ATLAS Test Program Synthesis tool has not yet been developed.

Figure 26 illustrates the inputs and outputs of the C/ATLAS Test Program Synthesis tool. This tool should extract the Stimulus-Response and Response-Response temporal requirements from the Requirements Model and the individual test scenarios from the output files of the scenario generation algorithms and then execute the algorithm described in section 5.3.2. The temporal requirements may be captured in a manner similar to the requirement identifiers in the description blocks of the SCR model.

6.4 Experimental Frame Synthesis

Once the C/ATLAS test programs have been generated, they may be applied at various levels during the design process. One essential level is at the virtual level where design models are to be simulated and evaluated as part of the Model-based Codesign process. Recall that at this level, *Experimental Frames* as described in chapter 1, are coupled with the design models to define and control the execution of the simulations.

There are at least two approaches to using the C/ATLAS test programs at the virtual level. One approach is to use these test programs as inputs to a suitable "generic" Experimental Frame that can interpret the C/ATLAS in order to apply the proper stimulus and collect the system responses. Another approach would be to convert the C/ATLAS

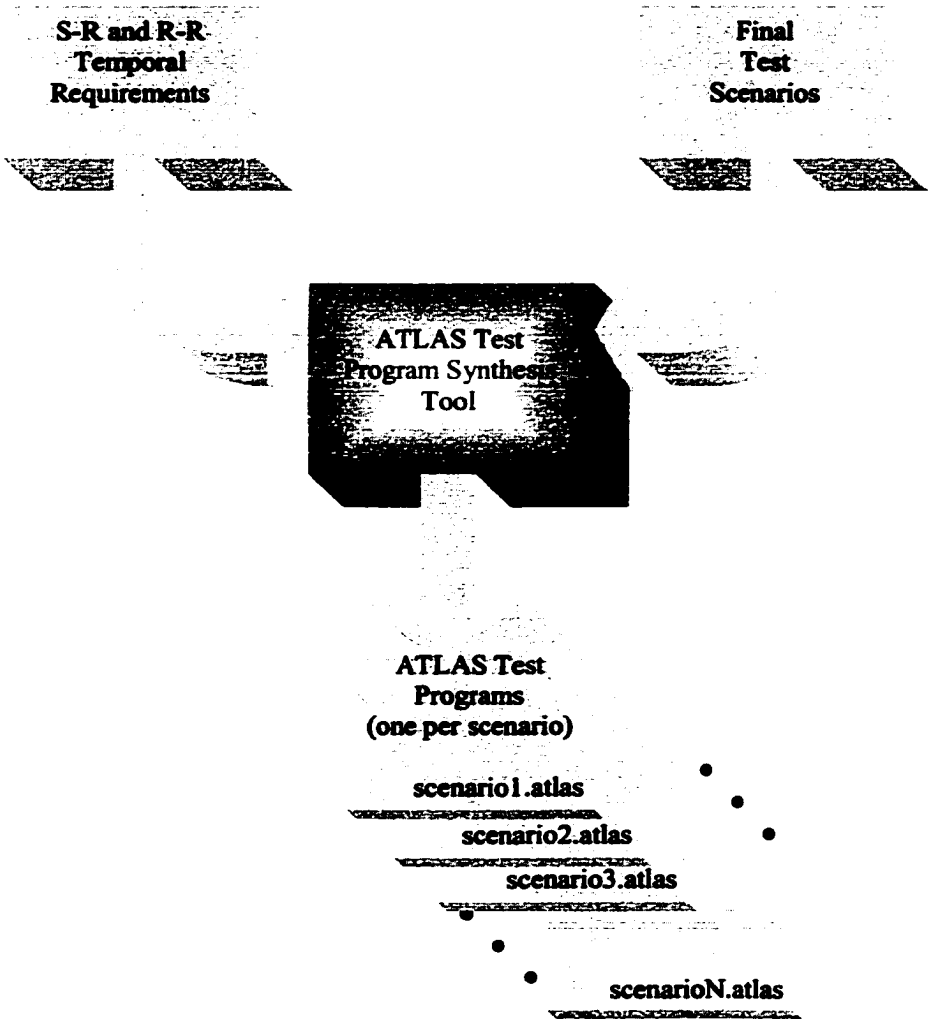


Figure 26. C/ATLAS Test Program Synthesis Tool

test programs into an equal number of experimental frames (e.g., synthesize java code for DEVS-java [69] Experimental Frame implementations). The former approach was selected for a prototype tool implemented by Jaroch [70, 71] to support testing design models developed in Statemate Magnum [16]. The point of application for this tool is identified in Figure 2 by the label ④.

The Experimental Frame developed by Jaroch is generic because a separate tool is used to interrogate the Statemate design model (with error checking against the C/ATLAS Test Program) and synthesize input and output interface modules. The code synthesis capabilities of Statemate Magnum are then used to allow the generic Experimental Frame core to be linked through the interface modules to any Statemate design model adhering to the interface specification. The generic Experimental Frame can then process the C/ATLAS test programs and interact with the design model through a standard set of functions. Figure 27 illustrates the arrangement of the generic Experimental Frame, interface modules, and model under test.

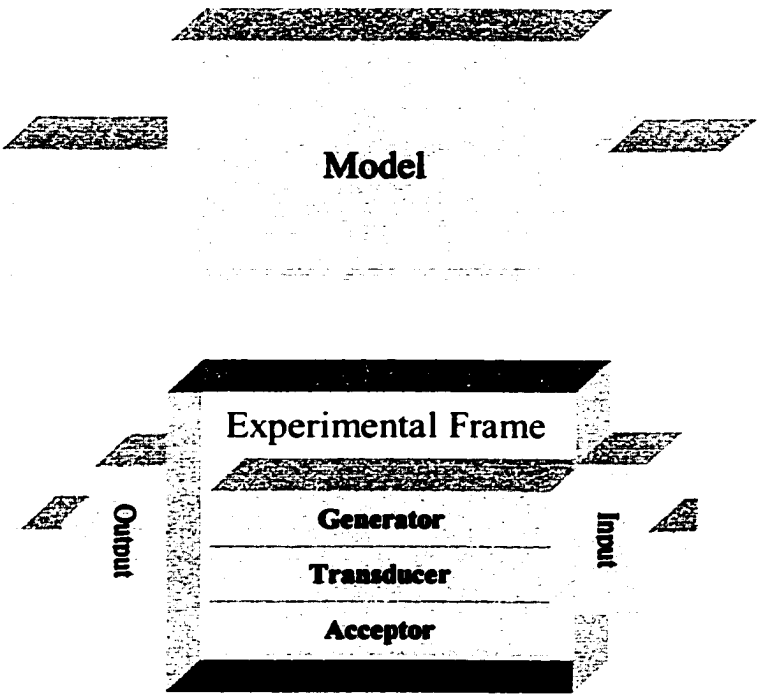


Figure 27. Generic Experimental Frame

6.5 Experimental Frame Translation

When a physical prototype of the system is available, the suite of experimental frames are intended to be used as the basis for the tests to be applied by a suitable real-time test environment (RTTE). Such a test environment may directly emulate the components of the Experimental frames [32, 33]. Alternatively, the RTTE may use the C/ATLAS test programs directly (not shown in Figure 2), which may eliminate the need for a translation effort. This latter approach may be more suitable if the Experimental Frames have been implemented in a generic manner [70, 71].

To assure that the physical system accurately reflects the intended design, it is of paramount importance that the applied tests be consistent (although of differing degrees of fidelity) across different levels of the design process. If the Experimental Frames are to be translated from the virtual to the physical level, automation of this task is perhaps the only way to guarantee this consistency between levels. Since each RTTE is expected to be unique, a set of tools, one for each target RTTE, would be needed. Although such tools have yet to be developed, the point of application is identified in Figure 2 by the label ⑤.

6.6 Execution of the C/ATLAS Test Programs

Execution of the C/ATLAS test programs may be performed in a virtual simulation environment or by a suitable RTTE. These steps are identified in Figure 2 by the label ⑥. The execution of the Experimental Frames should be performed as defined by the algorithm presented in section 5.3.3. The Experimental Frame synthesis tools developed by Jaroch [70, 71] supports execution of the C/ATLAS test programs at the virtual level,

although not exactly as defined by section 5.3.3. The notable limitation is that interpretation of R-S and S-S temporal requirements to allow adaptation to the actual responses of the model under test is not supported. These Experimental Frames require that fixed time WAIT FOR statements exist in the C/ATLAS test programs which is not adaptive.

As described in section 5.3.3, the execution process takes the R-S and S-S temporal requirements and the C/ATLAS test programs as input, applies stimuli to the model or system under test while adapting to the temporal realities of the test subject to ensure that the environmental temporal requirements are not violated, and logs time stamped stimuli and responses as output. This context is graphically illustrated in Figure 28.

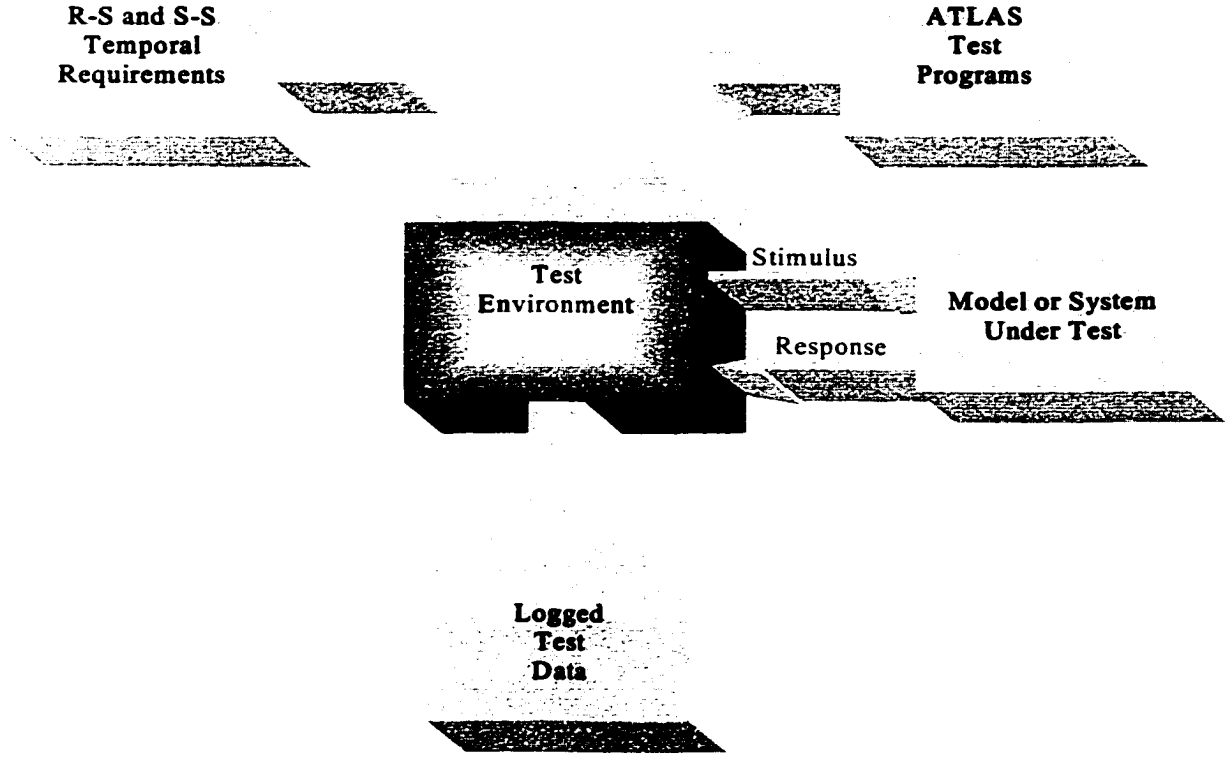


Figure 28. Application of C/ATLAS Test Programs

6.7 Analysis of Test Data

Analysis of the collected test data, identified in Figure 2 by the label ⑦, would certainly be tedious and error prone if left as a manual task. Although a prototype tool to automate this task has not yet been developed, such a tool should implement the algorithm described in section 5.4. This tool will take the logged test data, the temporal requirements that apply to the system, and either the final test scenarios or the C/ATLAS test programs as input. It will then compare the expected responses to the actual responses and verify the temporal requirements of the system to produce a test report. Figure 29 illustrates the input-output relationship for the Test Data Analysis Tool.

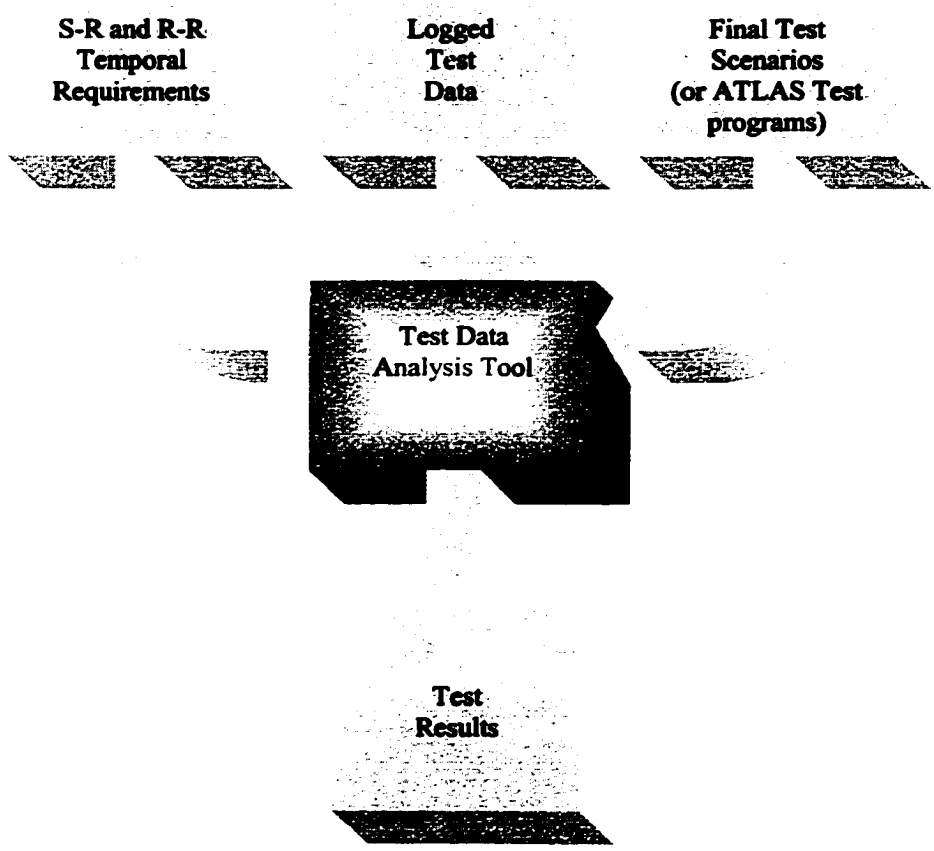


Figure 29. Test Data Analysis Tool

It should be pointed out that a practical goal would be to use the Test Data Analysis tool to analyze data produced from various levels of testing. To allow this to occur, the format of the logged test data must be specified and each test environment must produce test logs that adhere to this specification.

6.8 Graph Visualization Utility for Scenario Search Trees

Quite often during the development of the scenario generation algorithms, it was necessary to draw the scenario search trees. This was primarily to allow for analysis while troubleshooting implementation errors and to support discussions during presentations of concepts and algorithms. As this can prove to be time consuming and tedious, even for small systems (i.e., small scenario search trees) when the task must be performed numerous times, a method of automation was desired.

A graph visualization tool developed at the University of Bremen called daVinci [72, 73, 74] was evaluated and determined to be suitable, and was freely available to institutions of learning (i.e., the right price). A utility was then developed, which is called ELTodaVinci for EdgeList To daVinci. This tool will extract the scenario search tree description, which is given in an edge list format toward the end of either the output of the scenario generation part 1 algorithm (sga.c producing the base scenarios) or of the scenario generation part 4 algorithm (sca.c producing the final scenarios). It uses the graph package library to produce a daVinci term representation of the tree. This term file may then be opened with the daVinci Graph Visualization tool for viewing, printing, or embedding the graphs in documents. Three of the graphs contained in chapter 4 are the

result of using the conversion utility and daVinci. The simple input-output relationship for this utility is illustrated in Figure 30.

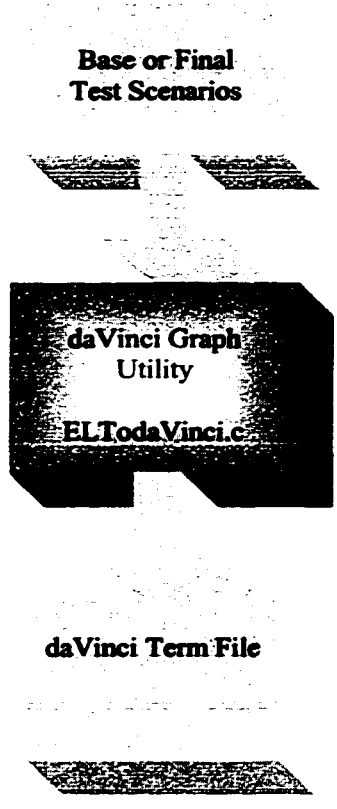


Figure 30. daVinci Graph Utility

7 EXPERIMENTAL RESULTS

This chapter will report on the results collected while experimenting with the implemented test generation algorithms. Experiences, observations, and comparison to other related work will also be discussed.

7.1 The Example Systems

Several systems have been used to evaluate the test generation process. The Safety Injection System (SIS) that has been used throughout the preceding chapters has actually been implemented in five variants. The version used in the examples is a version that was modified from the SIS more commonly presented [48, 49, 50] to ensure that the distance-based search portion of the scenario generation algorithm would be tested. The common SIS was implemented in four variants. The first two, called SISsm and SISlg, correspond to the common SIS and differ only in the allowed range for the water pressure input and thresholds as described in [45]. The remaining two variants of the SIS, SISsmNA and SISlgNA, are equivalent to the SISsm and SISlg models except that a set of negative action transitions have been added to allow for better comparison to the results presented in [45]. There were no textual requirements assumed for the common SIS. Unique requirement identifiers were simply assigned to each state transition in the model.

A Temperature controller system has also been implemented. This system can be thought of as a digital thermostat with a few added features and is described in [47]. This model differs from the others in that it was not implemented in SCR and then married with the scenario generation algorithms by synthesizing code. It was developed directly

in C from the requirements forms proposed in [47]. As a result, the structure of this model is somewhat different than those originating from an SCR model in that the modularity is based on procedure (altering more than one model variable) rather than on functions (modifying exactly one variable). The fact that this model works equally well in the scenario generation process helps to support the claim that any state-based modeling approach can be used to define the requirements models. All that is needed is that the model be translated into equivalent code along with interface and support code to allow for execution with the scenario generation algorithms.

The final system modeled is that of an elevator controller. This model represents a controller for an elevator in a three story building. The textual requirements for this model are included in Appendix H.

Table 16 below provides a summary of the models used and a few statistics about each.

Table 16. Summary of Modeled Systems

Name	SCR model	No. of Requirement IDs	Temporal Requirements	Statemate Model
SIS	Yes	11	Yes	Yes
SISsm	Yes	10	No	No
SISlg	Yes	10	No	No
SISsmNA	Yes	24	No	No
SISlgNA	Yes	24	No	No
Temperature Controller	No	31	Yes	No
Elevator Controller	Yes	63	Yes	Yes

7.2 Test Scenario Generation Results

The upper portion of the test generation process of Figure 2 has been applied to each of the models described in the preceding section. This portion takes us from textual requirements to a suite of test scenarios and, except for the development of the requirements model from the textual requirements, has been fully automated by the prototype tools described in chapter 6. The raw data for each graph presented in the sub-sections to follow is included in Appendix I.

7.2.1 Requirement Coverage by Algorithm

After having developed the two algorithm approach to generating the base scenarios, an interesting question is how many requirements were covered by each algorithm. As previously established, the greedy search is not a complete solution, but since every state added to the scenario tree covers at least one requirement ID, greedy search is expected to be more efficient than the distance-based search. An ideal situation is when all requirements are covered by the greedy algorithm.

Figure 31 illustrates the coverage by algorithm for each model tested. The coverage achieved by the greedy algorithm for the SIS, Temperature Controller, and Elevator Controller models seems reasonably good. The coverage for the greedy algorithm for the other variants of the Safety Injection System is not as good.

The reason for this apparent difference is an alteration to the test generation algorithms that was made to allow direct comparisons to the results reported by Gargantini and Heitmeyer in [45]. The approach taken in [45] includes a provision to

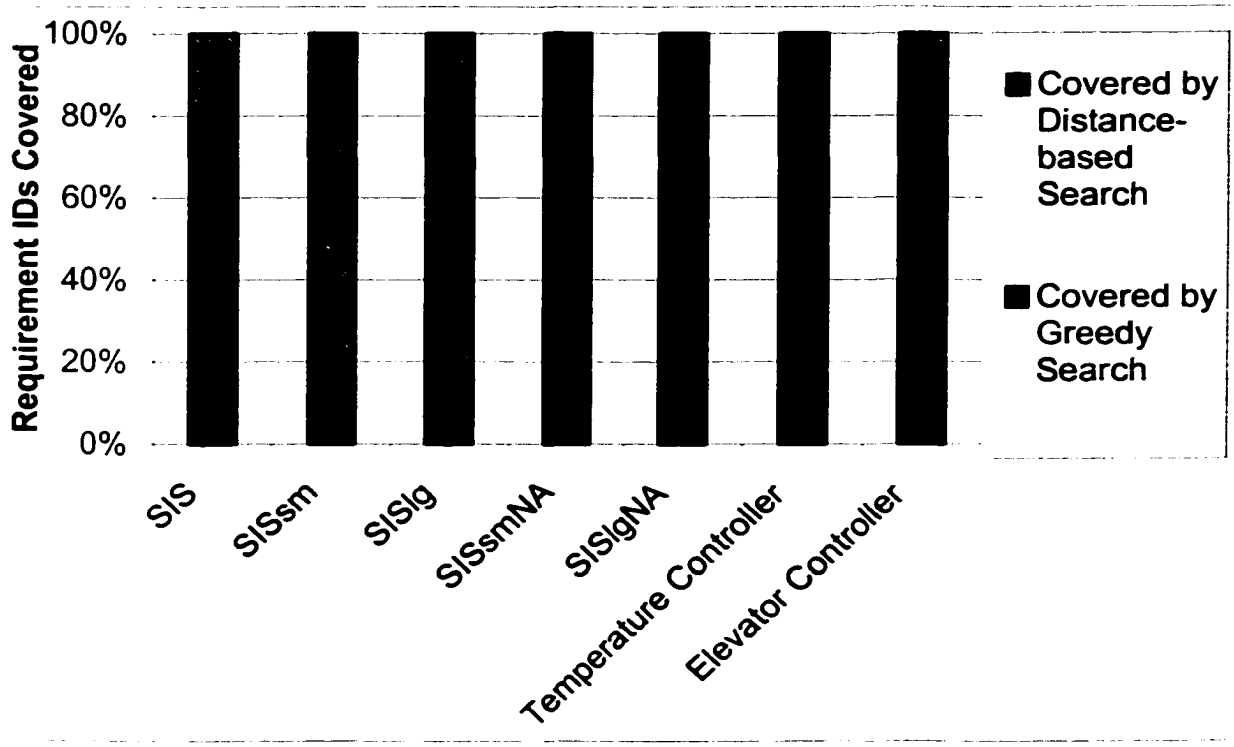


Figure 31. Coverage by Algorithm

accurately model the reality of physical inputs to the system. For the SISsm model, the *WaterPres* was constrained to change by a step size of not more than 3 pounds per square inch. Arguments can be made as to whether this provision is necessary or even desirable for test generation. It may be good in the sense that test scenario stimuli are kept within the expected normal rate of change for the environment. On the other hand, testing unrealistic inputs may be important to testing for high reliability systems. For the modeled Safety Injection System and similar systems were it makes no difference to the system whether the inputs respect physics or not, modeling such aspects only increases the test generation effort and the length of the resulting scenarios. For systems that

include provisions to detect inputs deemed abnormal in some sense, it may actually be necessary to include the provision to model both normal and abnormal input behavior. For example, if a system detects an abnormal input change, it may enter an error recovery mode. For these types of systems, the generated test scenarios must include both types of input changes in order to adequately test the system.

For the scenario generation approach presented in this work, abnormal and normal input changes can be modeled by listing absolute or delta test values in the input parameter file as described in section 6.1.1. Also, specifying an input to represent part of the system state allows the scenario generation algorithms to continue to explore even when the inputs are constrained such that the changes do not truly effect the state of the system. This was done for the common Safety Injection System variants where changes to *WaterPres* were constrained. This resulted in the poor coverage of the greedy algorithm. When absolute test values are specified for the common SIS variants, the greedy algorithm is able to cover 90% of the requirement identifiers for the SISsm and SISlg systems and 67% of the requirements identifiers for the SISsmNA and SISlgNA systems.

7.2.2 Requirements Verifiable at the Black Box Level

With the emphasis on the need to support testing at the black box level, the number of requirements verifiable at this level must be evaluated for the example systems. Figure 32 illustrates the performance of the base scenarios and the scenario enhancements in this regard.

All requirements are verifiable at the black box level for the SIS and Elevator Controller models. There is one requirement identifier common to both the SISsm and SISlg systems that remains unverifiable. This requirement identifier represents a redundancy in the common Safety Injection Models. After the Scenario Enhancing Algorithm failed to generate an enhancement for this requirement identifier, it was a simple matter to determine that this transition, which sets the internal variable *Overridden* to False when *WaterPressure* rises above the threshold HIGH, is redundant. The reason is that another transition sets *Overridden* to False when *WaterPressure* once again falls below HIGH. Of course this redundancy does not effect the black box behavior of the system and would not hurt in an implementation (other than being unnecessary computation), but it should not exist in a requirements specification.

The SISsmNA and SISlgNA systems, by definition, contain multiple requirement identifiers that do not change the state of the system. The Temperature Controller System includes a requirement that the system should ignore certain inputs when the system is in the *idle* mode. This translated into two requirement IDs in the requirements model. Because the Scenario Enhancing Algorithm is based on propagating a state difference until observable at a system output, this process will obviously fail if there is no difference to start with.

The discussion in the preceding paragraphs indicate that it is not always possible for the test generation algorithms to generate tests that are able to verify the correct functionality at black box level. The two cases where this is not possible is when the

assumption that the requirements model is minimal has been violated and for *negative action* requirements.

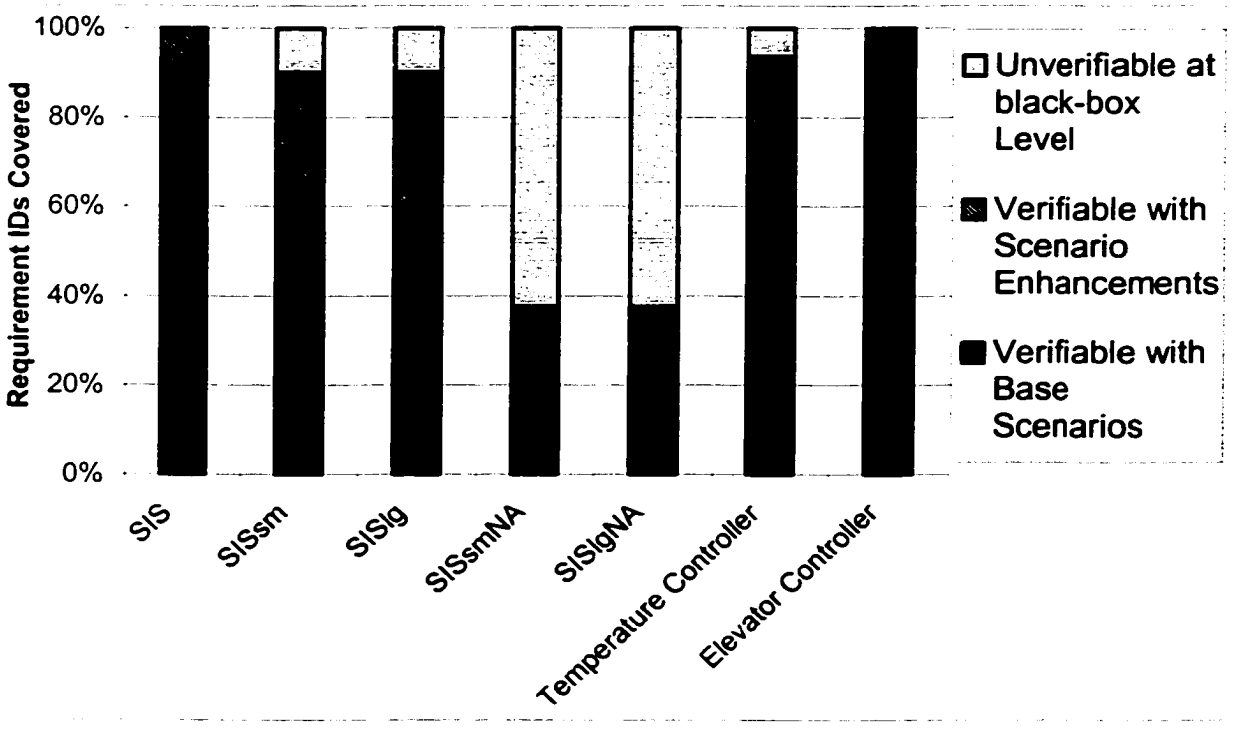


Figure 32. Black Box Verifiable Requirements

7.2.3 Length of Generated Scenarios

The cost of testing makes shorter test suites more valuable than longer ones, assuming equal fault coverage. Figure 33 provides the measured data related to test length for the example systems. This data only reports on the length of the test scenarios, counted by the number of inputs applied to the system and does not include the temporal aspects. The temporal requirements will effect test time and would provide a better measure of the true cost of the test suite. However, the temporal requirements are system

dependent and generally cannot be manipulated by the test generation process. For this reason, the lengths as reported here should provide an adequate measure of the quality of the test scenarios.

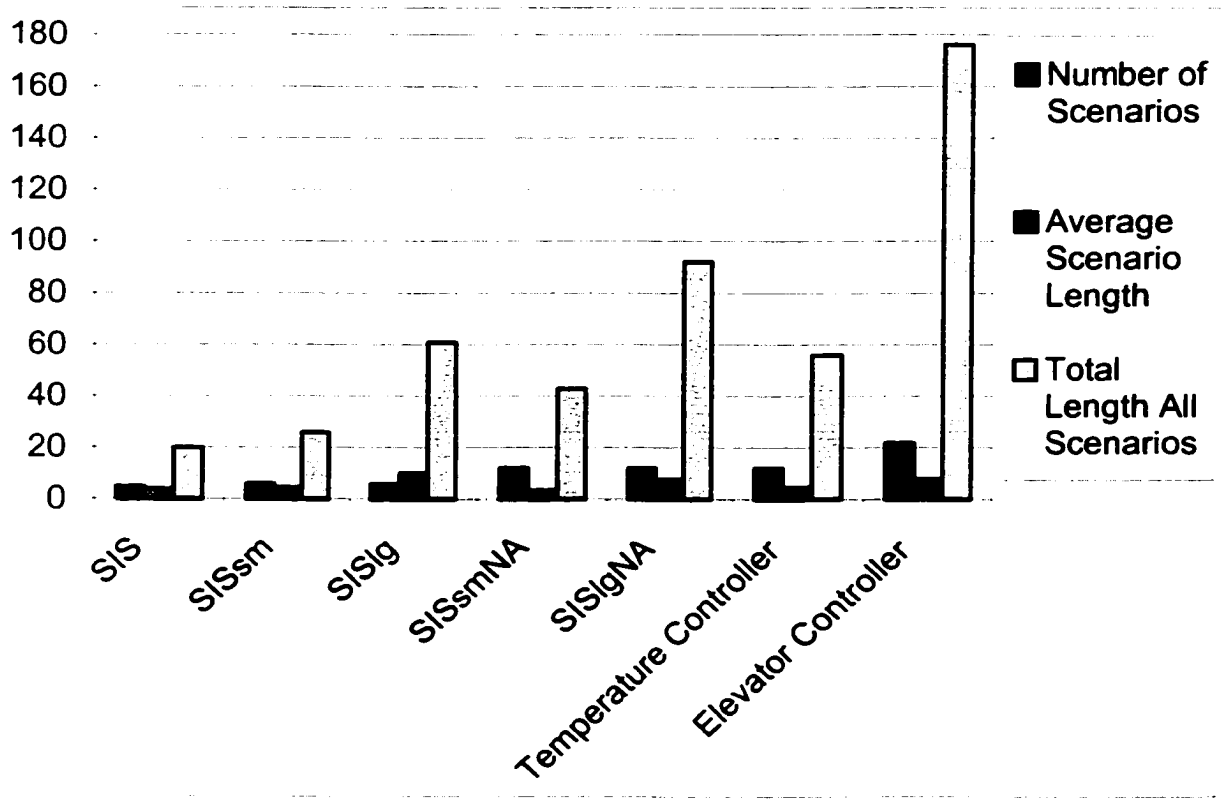


Figure 33. Length of Generated Scenarios (restricted *Waterpres* changes)

There are three values plotted for each model in Figure 33, the number of scenarios, the average scenario length, and the total test length. The data shows that the number of scenarios and the average scenario length grow very slowly as the models become more complex. The total test length grows more rapidly. Unfortunately, the purity of this data has been tainted because not all models have been subject to the same set of rules. Recall that the SISsm, SISIg, SISsmNA, and SISIgNA models had the *WaterPres* input

constrained to change in small increments, while the others have not. This directly effects the length of the generated test scenarios.

In order to enable interpretation of the scenario length data, test scenarios where regenerated for the SISxx and SISxxNA models without the constraints on the changes to *WaterPres*. This data is plotted in Figure 34. Postulating a simple relationship between the number of requirement identifiers and the scenario length measures, the horizontal axis has been changed to the number of requirement identifiers. For the measured data, this relationship appeared to be essentially linear for the number of scenarios and the average scenario length, so best fit lines were added to the chart[3]. A 2nd order polynomial provided the best fit for the total test length. Although, additional data on larger models is certainly needed to support these apparent relationships, they bode well for scalability in terms of the practical usefulness of the generated test scenarios.

7.2.4 Search Efficiency

Because the generation of the base scenario uses heuristic search algorithms, an interesting question is how efficiently these algorithms are finding useful state transitions that cover requirements. The output of the scenario generation algorithms includes the

---

[3] The curve fits in all figures in this chapter were calculated by MicroSoft Excel. The fitness measure, R^2, is defined to be $R^2 = 1 - \frac{SSE}{SST}$ where $SSE = \sum \left(Y_i - \hat{Y}_i\right)^2$ and $SST = \left(\sum Y_i^2\right) - \frac{\left(\sum Y_i\right)^2}{n}$.

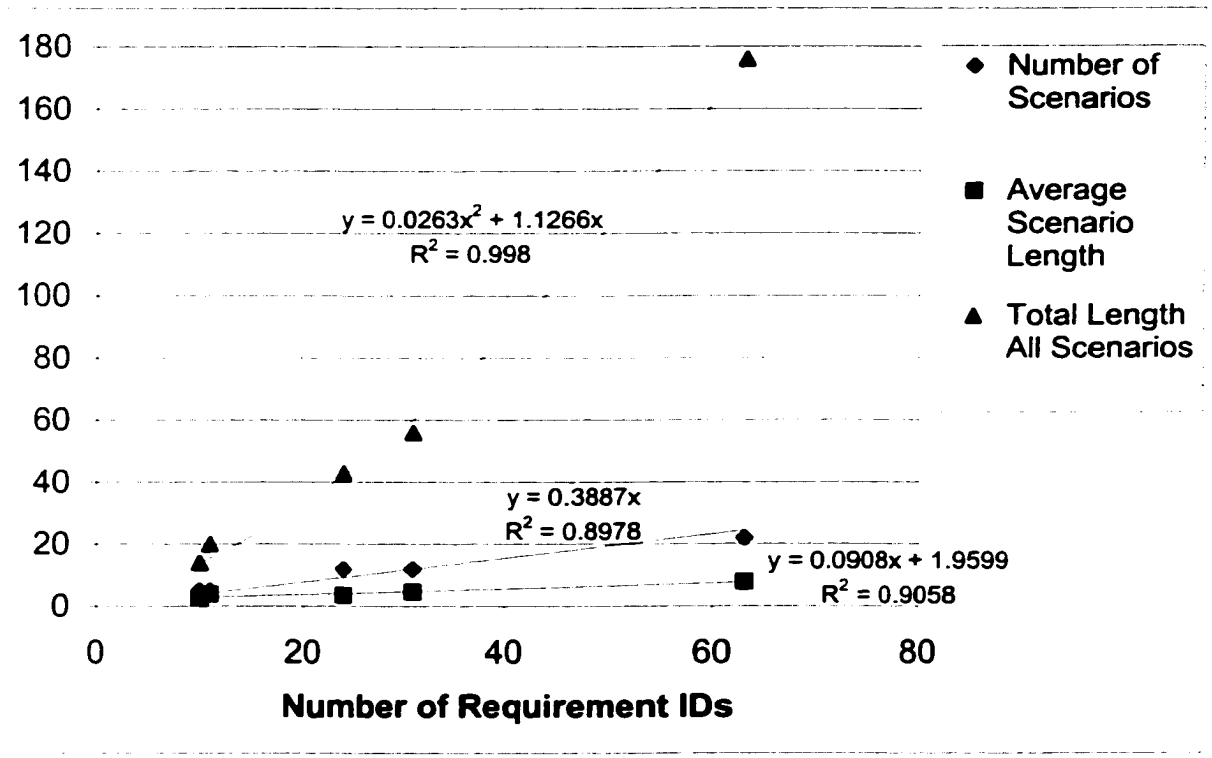


Figure 34. Length of Generated Scenarios (unrestricted *Waterpres* changes)

total number of states reached by all scenarios, which is the number of states in the scenario tree (*ST*). Also reported is the complete scenario search tree (*SST*). The *SST* includes all states explored during that portion of the test generation process. Ideally, the number of states in the *ST* and *SST* would be equal, which would represent the case were no time was spent expanding unneeded state space. This measure of waste will also be reflected, though less directly, in the run time measurements presented an a subsequent section.

There are two areas for this measurement, the generation of the base scenarios and the generation of the scenario enhancements. Recall that the generation of the base scenarios includes the greedy and distance-based search while the generation of scenario

enhancements uses the difference-based search. Figure 35 displays the percentage of the total state space explored that resulted in useful test scenarios. This data shows that the search efficiency is, in general, inversely proportional to the size of the system with some variation due to the nature of each system. The data for the SISxx and SISxxNA systems represents the case where changes to the *WaterPres* input were not restricted.

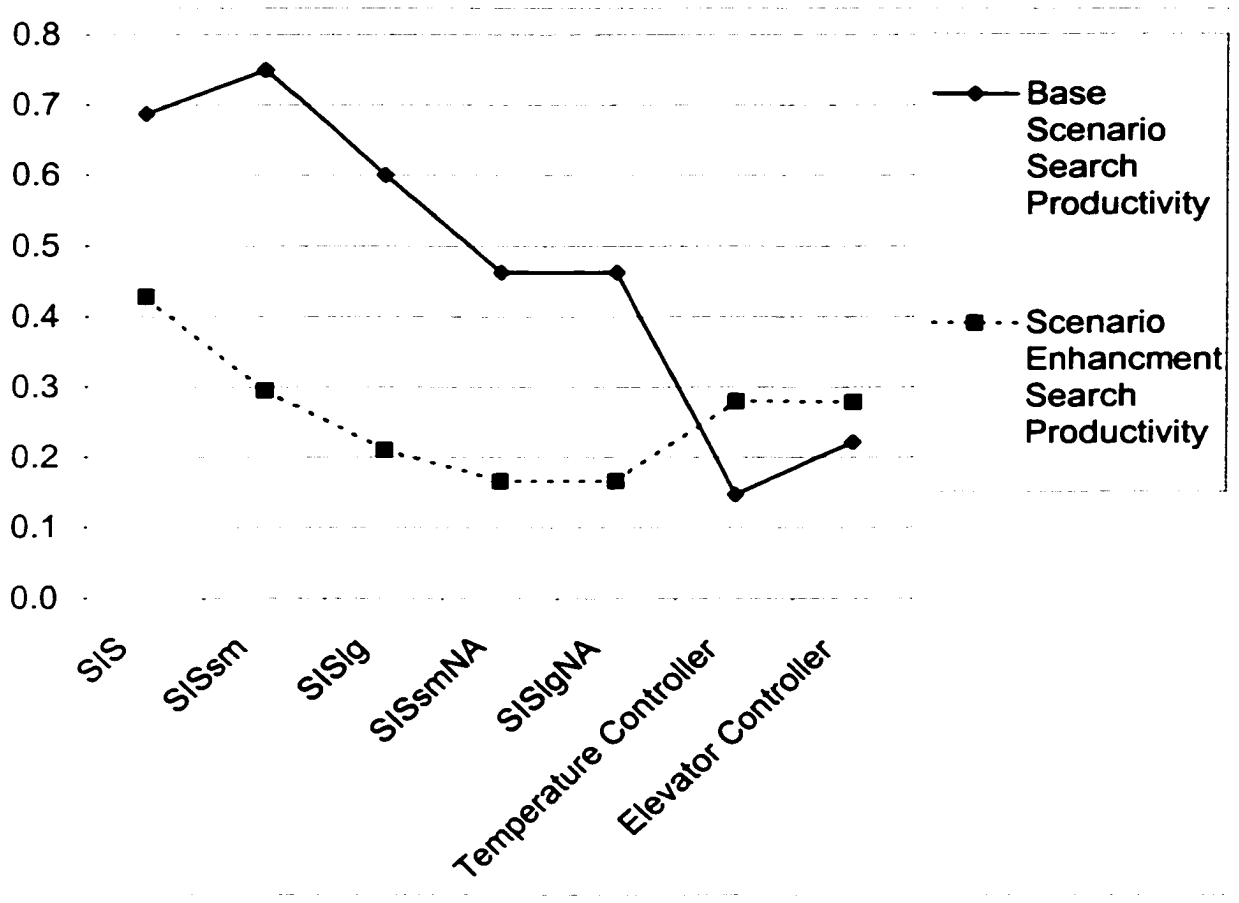


Figure 35. Search Efficiency: ST Vs. SST

7.2.5 Test Suite Efficiency

After generating a complete suite of test scenarios, one might wonder how efficient the suite is. Efficiency is measured in terms of the number of requirements covered vs. the total test length. Because system level tests will necessarily have a certain amount of overlap or redundancy in any test suite, due to the need to start each test from the system reset state, it may be beneficial to measure this redundancy as well. Because the redundancy is tightly coupled to the behavior of the system (i.e., a parameter not a function of the test generation method), and the amount of redundancy adversely effects the efficiency of any test suite, these measures should be considered together.

Test suite redundancy will be defined as

$$\sum[L_{shared} \bullet (N_{shared} - 1)]$$

where L_{shared} is the length of a shared sequence and N_{shared} is the number of times that sequence is shared between all pairs of test scenarios. Shared sequences will not be counted if the sequence is a subsequence of a larger shared sequence. A shared subsequence is any matching sequence of <start state, Stimuli/Response, end state> triples. For instance, suppose a test suite contains four test scenarios, A, B, C, and D. Scenario A contains sequence $X = p, q, r, s$ (where the symbols represent matching triples) which is shared with scenarios B and C. Scenario D contains sequence $Y = p, q$. Scenarios A, B, and C share a sequence of length 4, and scenarios A, B, C, and D share a subsequence of length 2. Any one of scenarios A, B, or C will provide coverage of sequence X and the others can be thought of as repeating this sequence, hence this sequence is repeated $3 - 1 = 2$ times. Although scenario D shares sequence Y with A, B,

and C, the occurrences in two of these latter scenarios will not be counted because they have already been counted in the larger shared sequence X. Hence, sequence Y is repeated $(4 - 2 - 1) = 1$ times. The total redundancy for this test suite is then computed as $4 \bullet (3 - 1) + 2 \bullet (4 - 2 - 1) = 10$.

Because the generated test scenarios are derived from a rooted tree, the repeated sequences will exist solely in the form of shared prefixes. This greatly simplifies the calculation of the redundancy factor. Figure 36 plots the raw test suite efficiency with the solid line, where again the rate of change for *WaterPres* in the SISxx and SISxxNA models was not constrained. The dashed line is a compensated efficiency which has the redundancy factor as defined above subtracted from the total test length. This compensated efficiency shows that the generated test suites are testing close to one requirement per stimuli. The uncompensated efficiency remains above one half for all the example systems except the Elevator Controller, which indicates that test length is approximately twice as large as the number of requirement identifiers in the requirements model.

7.2.6 Execution Times

Execution time is an important measure because the time needed to generate a suite of test scenarios limits the practical size of systems to which this approach can be applied. Of course faster is better, but an execution time requiring that test generation be left to execute overnight or over a weekend would be acceptable in many industrial settings.

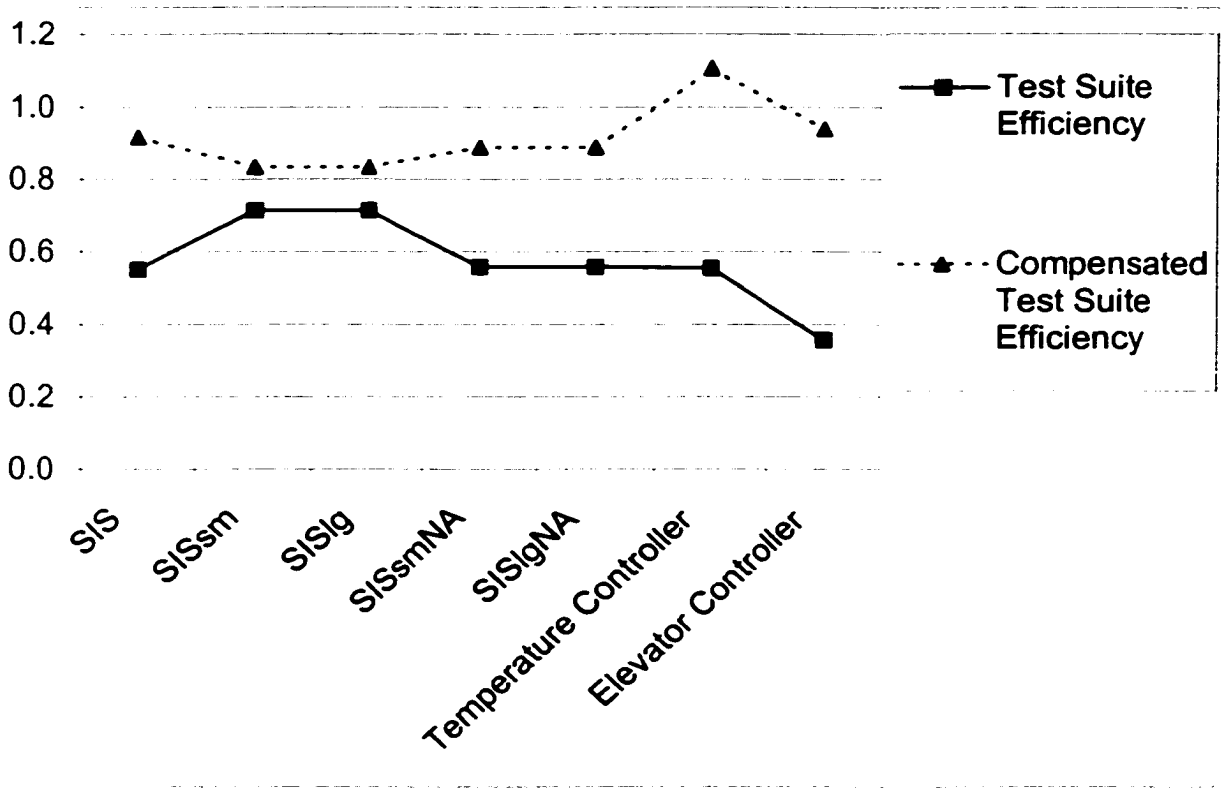


Figure 36. Test Suite Efficiency

The reported execution times presented in this section were measured using the standard C library clock() function. Test generation in all cases was performed on a very lightly loaded Sun Sparc Ultra-1 workstation with 128 MB of RAM running Solaris 2.5.1. The resolution of the clock() function on this platform is 10 milliseconds.

Figure 37 illustrates the absolute execution times for each of the four scenario generation algorithms. Note that reported times of zero indicate that the execution time was less than the 10 milliseconds timer resolution. These plots indicate that the execution times grew rather slowly with increasing model complexity, with the exception of the Elevator Controller. Although the Elevator Controller only contained about twice

as many requirement IDs than the Temperature Controller, the larger increase in execution time for the Elevator Controller can be attributed to another measure of complexity. The Elevator Controller requires longer sequences of inputs in order to take the model into the modes needed to exercise requirements (i.e. the model structure).

The poor execution time observed for the *sea* algorithm with the SISxxNA models is due to the fact that the *sea* algorithm will attempt to find enhancements for each of the negative action requirements, which are guaranteed to fail as previously discussed.

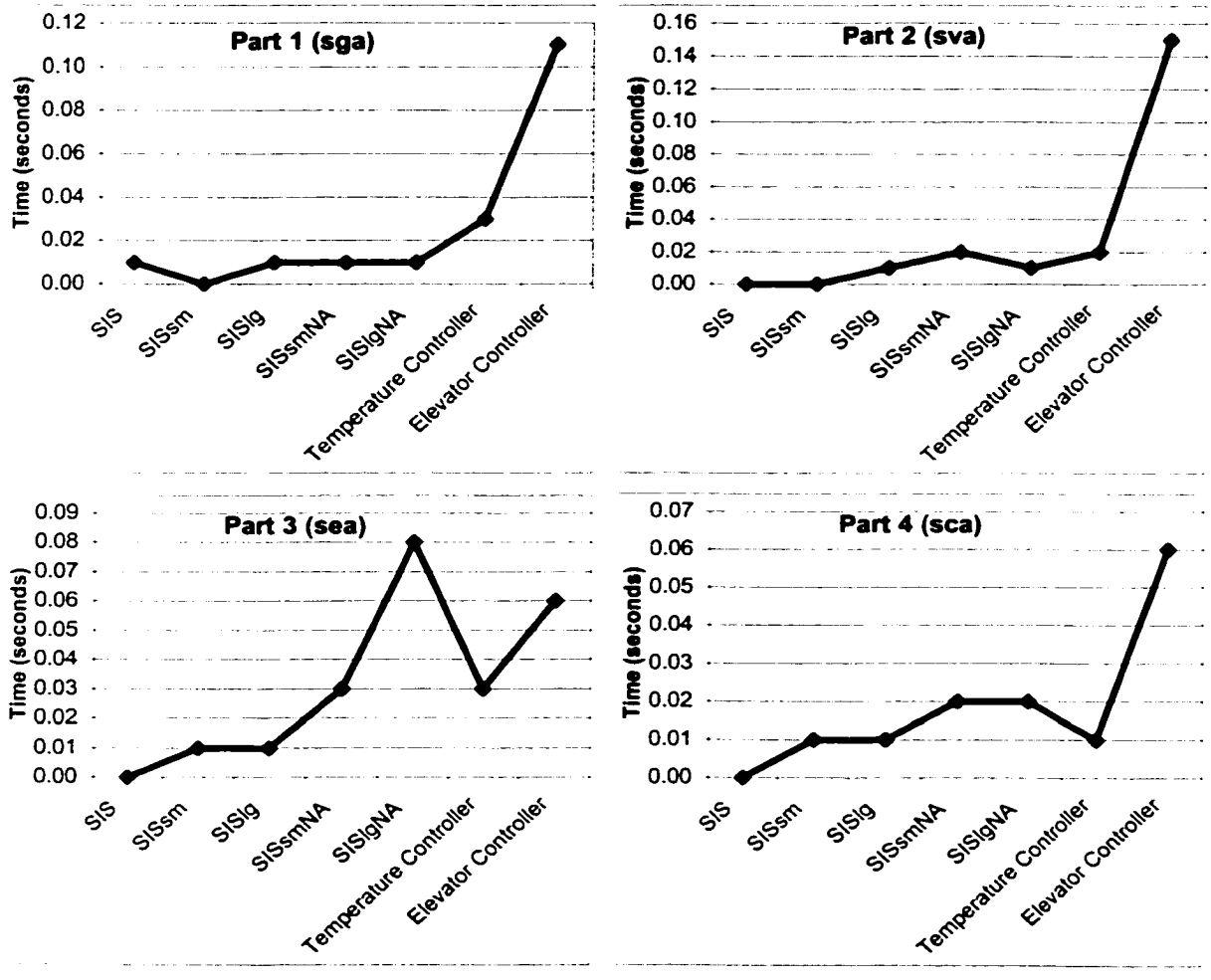


Figure 37. Absolute Execution times for SGA Algorithms

It is useful to compare relative execution times of a computation, particularly to determine were the majority of time is being spent in order to target optimization efforts. This practice is termed profiling. Figure 38 shows the relative execution times for the four scenario generation algorithms. The results for the SIS model are misleading due to a lack of timer resolution. The measured time for the *sga* was 10 milliseconds and 0 seconds was measured for the other three algorithms, leading to the apparent time domination by the *sga*. In reality the other algorithms executed in some amount of time between 0 and 10 milliseconds so the actual distribution of execution times was most certainly more balanced.

It is difficult to say that any one of the algorithms contributes significantly more to the overall execution time than any other. From this set of example systems, it does not appear that any one algorithm performs worse on average than any other. However, it does appear that model characteristics have differing effects on performance for each algorithm.

Because the *sga* and *sea* both involve search over a state space whose size is exponential in the number of state variables in the model (a reasonable measure of the size of the input), the worst case execution time of the combined four part scenario generation must also be an exponential relation to the size of the input. However, as is the case with some other algorithms that have a poor worst case performance but decent average case performance (i.e., least frequently used page replacement), it is believed that the scenario generation algorithms will also exhibit good average case performance.

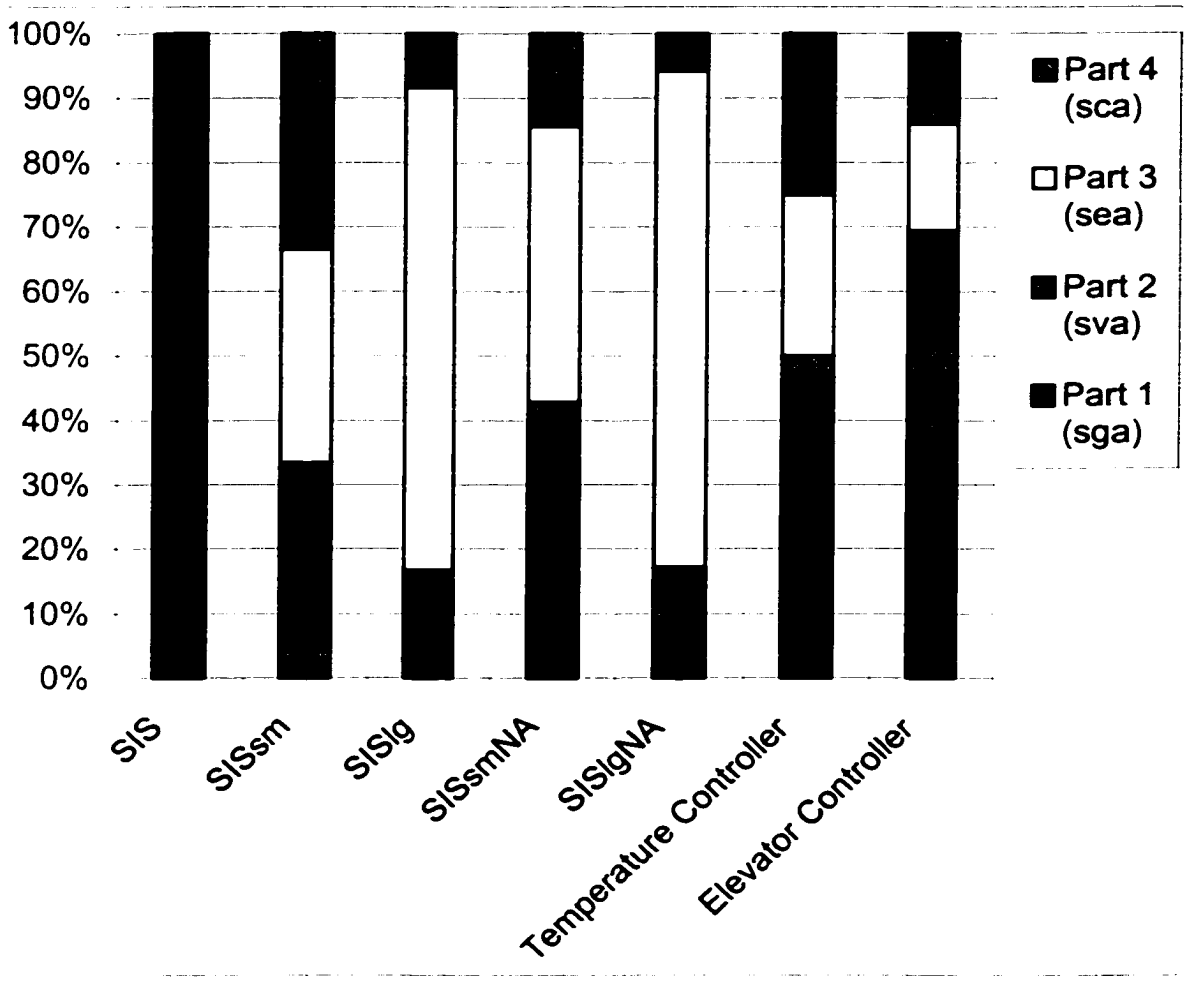


Figure 38. Relative Execution Times for SGA Algorithms

Using the total measured test scenario generation execution times for the example systems, we can attempt to empirically derive a relation between the size of the system and the time needed to generate test scenarios. Figure 39 charts the total execution times and a best fit curve. As was done for the modeled relation to the test suite lengths, the execution time measurements for the SISxx and SISxxNA models are for runs where the constraint on changes to *WaterPres* was not enforced. Fits with various curves were

attempted, such as power or exponential, with a 2nd order polynomial providing the best fit. This apparent relation is promising in terms of the ability to scale this scenario generation process to much larger systems. Based on this relation, it would take one hour to generate a suite of test scenarios for a requirements model with just over 7,700 requirement IDs. Measurements with some larger systems are still needed to increase the confidence in this execution time relation.

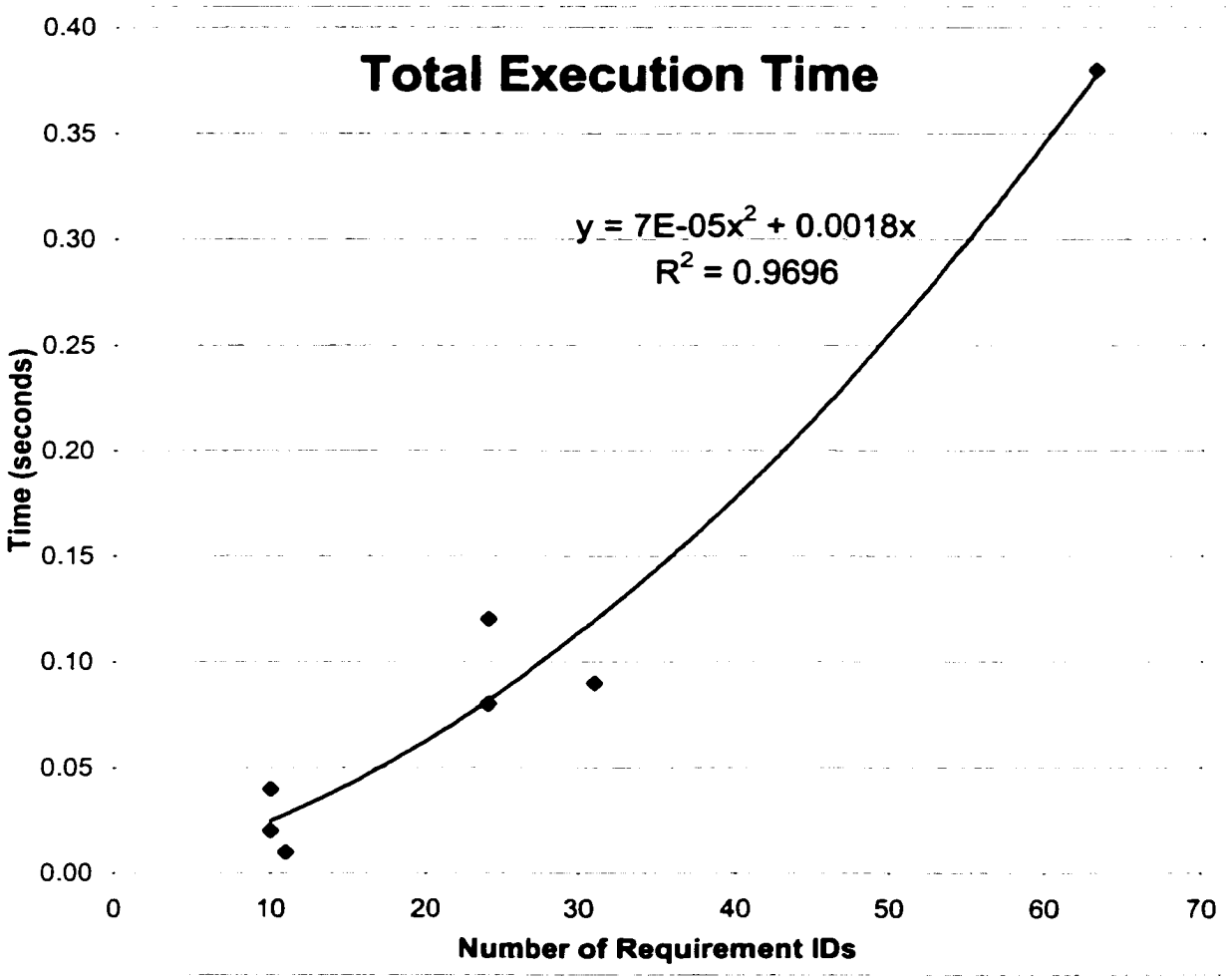


Figure 39. Total Execution Time

7.2.7 Binary Search Tree Statistics

Three of the scenario generation algorithm, *sga*, *sea*, and *sca* maintain a binary search tree (BST) of all unique states in the scenario tree. This is implemented as a second set of edges on the set of nodes in the scenario tree. The BST is created by inserting nodes as each unique state is generated during the execution of the algorithms. The use of a simple binary search tree is based on the assumption that the tree will be generated in a sufficiently random manner to produce a tree that is fairly well balanced. If this is not the case, a variant such as a Red-Black tree [52] could be used. Unfortunately, using a Red-Black tree includes added execution cost to rebalance the tree after each insertion.

Figure 40 plots the measured average path lengths for the generated binary search trees. Also plotted is the average path length for an optimally balanced BST. This result shows that the generated BSTs are reasonably well balanced and should remain within about 1.67 times the optimal value based on the logarithmic curve fit to the measured data. Hence the simple BST implementation appears to be adequate and the added expense of a Red-Black tree is not warranted.

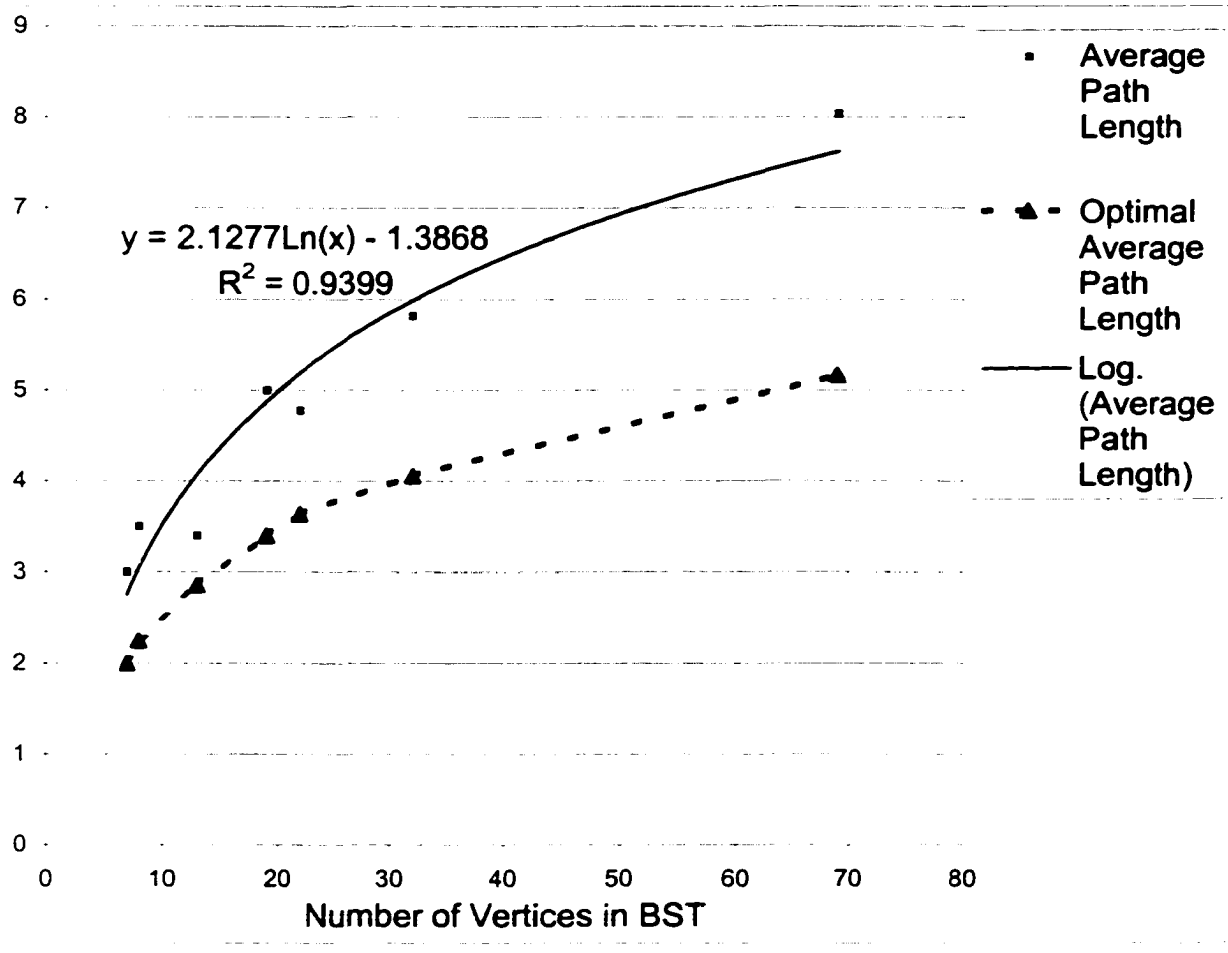


Figure 40. Binary Search Tree Average Path Length

8 CONCLUSIONS

This chapter is devoted to emphasizing the accomplishments, relationships to other works, and future of the research presented in this monograph. The first section summarizes the major contributions of the work presented in the preceding chapters. The second section summarizes the peer context by comparing and contrasting this work with related research. The final section discusses limitations and suggestions for future research that can build on the results of this work.

8.1 Summary of Results

The topic of testing computer based systems has been explored and found to vary widely in the goals, levels of application, and methods of generating test suites. This broad topic was narrowed by focusing on event-oriented real-time embedded systems and testing within the context of Model-based Codesign. Additionally, an argument was made that even with alternatives such as formal logic specifications and proof assertions, testing will always play an important role in the design of computer based systems.

The primary research goal was defined to be the development of provide a method to automatically generate a suite of test scenarios from the requirements for the system. The goal of the generated suite of test scenarios is to allow the system design to be validated against the requirements. The topic of requirements capture and documentation has been explored in the context of how the requirements can be used to support the research goal. The benefits of automatic test generation include improved efficiency, completeness (coverage), and objectivity.

The Model-based Codesign method has been refined by defining a design process flow. This process flow includes the generation of test suites from requirements and the application of these tests across multiple levels of the design path. Testing in the context of Model-based Codesign for real-time embedded systems places special emphasis on generating test scenarios that allow for black box testing and testing of temporal requirements.

The difficulty of the problem of automatic generation of optimal test scenarios, even for event-oriented systems, has been established, resulting in the pursuit of heuristic solutions to the problem. An approach has been proposed that utilizes what has been called a requirements model and a set of four algorithms. The scenario generation process made up of the four algorithms is the primary contribution of this research. The requirements model is an executable model of the proposed system defined in a deterministic state-based modeling formalism. Each action in the requirements model that changes the state of the model is identified with a unique requirement identifier.

A set of base scenarios are generated by applying a two phase algorithm with the goal of producing a set of scenarios that cover every requirement identifier in the model (i.e., exercises every state changing action). The first phase of the algorithm applies a greedy search heuristic that continues as long as additional requirement identifiers can be covered through the application of a single additional stimulus to the model. After the greedy search can no longer make progress, and second heuristic is applied. The second heuristic, termed the distance-based search, selects a requirement identifier remaining to be covered as a target, determines the model state enabling the state changing actions

associated with the target, and orders the set of potential states based on the distance to this enabling state. The potential state with the smallest distance is selected for expansion.

It has been demonstrated that the set of base scenarios is not adequate to test at the black box level. An algorithm to determine which state transitions are verifiable by applying the base scenarios and observing only the system outputs has been defined. This algorithm simulates the correct requirements model and the same model with the action associated with requirement identifier in question disabled in order to determine if a difference is detectable at a system output. If no difference is detected for any scenario covering a particular requirement identifier, then that identifier is listed as not verifiable at the black box level.

With the goal of enhancing the set of base scenarios to support black box level testing, the issue of determining whether the system has achieved the proper state by observing only system outputs has been researched. To this end, works related to state identification in state based systems have been explored. An approach related to fault propagation in combinational logic test generation has been selected as a heuristic solution to enhance the set of base scenarios. This heuristic algorithm, termed the difference-based search, targets a single requirement identifier not verifiable by the base scenarios at the black box level and simulates two versions of the requirements model in parallel. One model is the correct requirements model and the other is the same model with the state transition responsible for covering the target requirement identifier disabled, similar to the algorithm for determining black box level verifiability. Search

continues until a difference is observable at a system output or until no differences remain in any pair of expanded model states.

The final step in the test scenario generation process is to combine the base scenarios with the scenario enhancements. The algorithm defined to accomplish this task is relatively simple but must handle cases of overlapping scenario enhancements.

Another contribution of this research is the development of the process defining the generation and use of the test scenarios. This process also includes the treatment of temporal requirements which are considered separately from the generation of the test scenarios. Temporal requirements are treated as constraints on the functional correctness of the system. An algorithm has been defined to combine the test scenarios with the environmental temporal requirements to produce timed test scenarios in the IEEE standard C/ATLAS test language. An algorithm has also been defined to describe the behavior of the test environment as it interprets and applies the C/ATLAS test programs. The test environment must maintain adherence to the environmental temporal requirements and in order to do so must adapt to the responses of the system under test. Finally, an algorithm to analyze the test results logged while applying the test scenarios has been defined. This algorithm will verify the proper behavior of the system by checking the system responses and the timeliness of these responses.

Prototype Implementations of the scenario generation algorithms have been developed in the C programming language. These algorithms are supported by the prototype Requirements Model Code Synthesis tool. This tool synthesizes C code from requirements models developed in the SCR formalism using the SCR Toolset.

Having developed prototype tool support for the generation of test scenarios from SCR requirements models, it was possible to collect measurements on several interesting metrics of the scenario generation process. Although these measurements were made on a limited set of relatively small systems, the results support the position that the algorithms are performing reasonably well, that the generated test scenarios are adequately efficient, and that the processing time needed for test generation grows slowly enough to support much larger systems.

8.2 Summary of Related Work

This section provides a summary of similar research and of research in related areas that can be used to support automatic test generation from system requirements. Several categories have been defined to group the cited research. In some cases, references apply to more than one category. In these cases, the category that appeared to be most appropriate was selected.

8.2.1 System Analysis and Design

Our own Model-based Codesign method is certainly not the only approach to system design. A good overview and philosophical discussions regarding the analysis and design of systems is provided in [75].

One alternative is the ESCAM method. A case study using this method is presented in [76]. Further references are cited in [76] for detailed descriptions of the ESCAM method.

Another alternative design method using OMT and Fusion is given in [77]. The authors obviously have real world experience in software design. Chapter 3 is devoted to development of a requirements specification. The method presented uses Use Cases, Use Case Diagrams to show the hierarchy of Use Cases, and System Context Diagrams which are used to show the functionality of the system, the actors, and the environment of the system.

A method having many similarities to Model-based Codesign is furnished by Lovengreen et al., in [78]. Interface definitions and event sequences are integral parts of the presented method which exists in the domain of software engineering.

8.2.2 General Issues Related to Requirements Engineering

The root of the work presented here, and any well conceived system design effort, are the requirements. Over the past two decades formal treatment of requirements has emerged as a separate engineering discipline. As with any discipline, the domain, the goals, and the tools and methods used are continuously evolving.

White and Lavi provide a summary of a 1985 workshop on embedded system requirements in [79]. This report identifies unique properties of embedded systems that make using methods and tools developed for information processing systems unsuitable. It was reported that attendees agreed that a model of the proposed system should include some form of state machine, but that in order to control complexity, some structuring/partitioning based on the system environment is necessary. Recommendations for standards, tool development, embedded computer system models, methods, and tool support were then given.

In the early to mid '90s, Hsia, Davis, and Kung present the state of the art, issues, and suggested research directions in [14]. In [15] Davis and Hsia team up again to discuss issues related to the discipline and provide a *must* read list for requirements engineering. Siddiqi challenges the long held belief that requirements must only specify what must be done and not how it is to be accomplished in [80].

Finally, Davis [81] provides an extremely comprehensive reference for all works related to Requirements Engineering. In addition to a good treatment of the practice of managing software requirements, this text contains references to what Davis believes to be all publications related to requirements. These references include an abstract of each work and multiple indexes for quickly locating items of interest.

8.2.3 Requirement Capture and Analysis

In order to produce a quality requirements document or requirements model, suitable methods must be employed. During and after the requirement capturing process, the requirements must be analyzed to determine such quality measures as lack of ambiguity, completeness, and consistency. This is a large area of research and what follows is a sampling of work in this area.

Potts et al., report on a study from the point of view of cognitive psychology into the best methods for determining requirements through interactions with the customer in [82]. They report that the use of scenarios is critical to their requirements analysis process and discuss ways to represent scenarios.

Hooper and Hsia describe the three main facets of requirements engineering in [83]. These facets are identification, analysis, and communication. They also propose the use of scenarios as a means to produce a rapid prototype of the proposed system.

Ludewig and Boveri contribute a comprehensive work in [84] that discusses the goals of requirements in the context of the design process before going on to describe their requirement specification language. The authors use an interesting stalactite-stalagmite metaphor for the development process. Implementation is at the floor, development tools (synthesis, compilers, etc.) are the stalagmite while customers wishes are at the ceiling and requirements specifications are our attempt to create the stalactite. The goal is to get the two to meet. The desire for "what" not "how" is stated but then reasons are given as to why it is not possible to create requirements specifications for real systems that are purely "what." They conclude by describing their specification language, ESPRESO. ESPRESO appears to have many attributes of an object-oriented language, which is noteworthy because this work was done before the popularity of OO languages.

Miller and Taylor propose a method based on finite state machine models with structural decomposition in [85]. This proposal "crosses the line" and puts design dependent information (the decomposition) in the requirements. This is done to manage complexity. Background on black box descriptions and thier proposed "clear box" description are given. The method is outlined and tools are proposed to support it. SDL is proposed for the FSM descriptions. The Problem Statement Language/Problem

Statement Analyzer (PSL/PSA) is proposed for the structural decomposition and the Requirements Language Processor [4] is proposed for the black box descriptions.

Matsumoto et al., propose in [86] that the software development process can be described by the progression from Requirements Model, to Control and Data Flow Model, to Program Structure Model, and the transformations between each. The requirements model consists of *what* (entities), Relationships between entities, and constraints. The requirements model describes three aspects of the system: system environments, dynamic behavior, and information processing.

Buescher and Wilkinson describe a method to model requirements for real-time systems in [87]. The authors state that a single technique was not found to be sufficient to capture necessary requirement details. They propose a three view perspective of system developed. The views are processing (identify all event-response pairs and assign a process to each pair), information (depicts the data within the system and the data communicated across the system interfaces), behavioral (describes how the system generates output signals from input signals and how signals affect system processing).

Jaffe et al., contribute a work [88] describing the criteria for analysis of software requirements stated in languages based on finite state machines.

Heitmeyer et al., describe the SCR formalism in [48]. This work includes the method for developing the requirements specification (model) in the SCR formalism and methods to automatically check the specification for a number of desirable properties. Bharadwaj and Heitmeyer [50] describe a method to convert SCR specifications to the Promela language and then use the Spin verifier to prove that proposed properties hold in

all reachable states. In [49] Heitmeyer extends the applicability of SCR by suggesting ways to deal with continuous variables and temporal constraints.

And to conclude this section, White gives a comparative analysis of eight requirements specification methods in [13]. Each of the methods is compared using a generic Entity Relationship (ER) model as the measuring stick. Detailed discussion of SA-RT, SCR, and Statemate are given in the paper with SCR comparing very favorably. A summary table is also included for the other languages.

8.2.4 Formal Requirements Languages

Levene and Mullery provide a overview of early requirements languages in [89]. This paper states the motivation for requirements languages and methods; omissions, poor presentation, ambiguousness, and inconsistency. This paper is promoting the use of computer tools to produce better requirements. The Software Requirements Engineering Methodology (SREM), Problem Statement Language/Problem Statement Analyzer (PSL/PSA), Software Development System (SDS), and System Encyclopedia Manager (SEM) languages are summarized. The results of an experiment using SDS is given. The conclusion is that the tools produce significantly better requirements. It is noted that some tools use fixed languages and some use languages that can be tailored to a particular domain. The fixed languages need to be sufficiently rich to be widely useful, and the definable languages must be managed and controlled closely in practice.

Demetrovics et al., describe Specification Meta-Systems in [90]. Meta-systems are specification systems that do not use a predefined language. Instead they are based upon general purpose methods and language definition facilities. Brief descriptions of the

PSL/PSA (Problem Statement Language/Problem Statement Analyzer), SDLA (System Descriptor and Logic Analyzer) and SEMS (System Encyclopedia Management System) are given. None of the methods in this paper appear to handle temporal or behavioral aspects of a system as they are geared toward the structural aspects of information systems.

Davis [4] presents the design of a family of application-oriented requirements languages. Application-Oriented means that each language is tailored for a particular application (domain). This is the "Specification Meta System" idea from [90]. Provides a good discussion of the trade-offs between formal and informal requirements and of strengths and weaknesses of some requirements languages such as the Problem Statement Language (PSL), Input-Output Requirements Language (IORL), and the Requirements Statements Language (RSL). Davis then defines the attributes of his application-oriented languages; system model (extended finite state machines), control structure common constructs, and four classes of timing constraints. Each language in the family should contain the same attributes but will be tailored in the use of vocabulary and relationships so that each is made easily readable by the users in the domain for which the language was tailored. Because of the commonality, all of the languages in the family can be processed by a set of common tools for static structural and behavioral inconsistencies.

In [91], Frinke et al., describe the Requirements Specification Language (RSL) developed for NASA. RSL is an object-oriented language designed to specify systems. The language seems to be geared toward the domain of information processing systems (e.g., database systems with transactions). The language provides constructs for object-

oriented (natural for transaction-oriented systems) and operation-oriented specifications (natural for transform-oriented systems). This language does not include a concept of time and does not address physical interface details. For these reasons, this language would be of limited usefulness for embedded systems. Some support tools do exist.

Cohen et al., [92] discuss a method to automatically monitor software to ensure compliance with requirements. The language used to capture the requirements is the Formal Language for Expressing Assumptions (FLEA). This is a run-time approach were the software is automatically monitored during testing or possibly during use.

In [6] and [17] Heitmeyer et al., describe tool support for the SCR language. A set of tools is available to support the development of SCR models, automatic consistency checking, symbolic simulation, and dependency graph browsing.

8.2.5 Test Generation methods

The paragraphs that follow give an overview of recent literature on automatic test generation. For the most part, the works are reported in chronological order.

Arguably the defining work on test generation from state-based specifications is by Chow [43]. Chow describes a method to test the control structure of software that can be modeled by finite state machines. This method is manual because executable specifications are not used. The three step method is to estimate the number states in the correct design, generate test sequences based on the current design, and verify the responses by comparing to the expected responses (derived from the system specification). The test sequences are made up of all sequences from root to pendent in the test tree. State distinguishing sequence are then added onto the end of each sequence.

These are guaranteed to exist based of the assumption of a minimal finite state machine representation of the system.

Two groups contemporary to the early work of Davis, used the finite state machine (FSM) description output of Davis' Requirements Language Processor (RLP) as the basis for test generation approaches. Bauer and Finger [41] augmented the FSM with system state information and expected system responses. From the augmented FSM a grammar is generated. From the defined grammar, sentences (i.e., scenarios) are created by generating valid strings of symbols in the language. Chandrasekharan et al., [40] also describes a specification and validation environment for real-time systems based on augmented FSMs from Davis' RLP. This work gives a good discussion of the limitations of using FSMs and why the author believes they are adequate for describing real-time systems. It also provides a good overview of the RLP semantic model. The test generation algorithm is given at a high level and involves edge-cover problems which are solved heuristically to give an $O(n^2)$ algorithm. Some experimental results are also presented.

Although Dasarathy [65] does not describe a test generation method, he provides a classification of temporal constraints and also discusses a testing method to verify expressions written in RTRL (Real-Time Requirements Language) and ATLAS . This method is shown to provide a test suite that is polynomial in the number stimuli and responses to be verified.

Hsia et al., provides a good treatment of scenario analysis in [9, 10, 18]. Conceptual state machines are derived from scenario trees. The state machines are verified against

the trees, then the state machines are used to exhaustively create all possible scenarios. A prototype is generated from the produced scenarios. Finally, all scenarios are validated by using the prototype. This approach does not support non-determinism or concurrency directly but multiple user views from which scenarios are created produce the possibility of having to deal with multiple concurrent scenarios in the validation step. The state of the practice is described by a good survey in [18].

Weber et al., [38] relate work done at Honeywell Technology Center toward automating the generation of V&V test artifacts for software testing in the domain of electronic flight instrumentation systems. Requirements are specified by a static object oriented model and a dynamic model which uses a Rule => Condition-Action paradigm. The test generator processes requirement rules and produces a test matrix which defines test inputs and expected outputs. Test inputs are vectors (i.e., single sets of input values). Sequential behavior is captured as notes at the bottom of the test matrix. Temporal requirements are not treated.

Frezza describes in [93] a set of relationships between requirements definitions (RD) data and architectural design (AD) data. These relationships are used to relate test sets that are applied to simulations of both the requirements and the design and to compare the results. The test sets appear to be manually generated. The domain is hardware design using VHDL. The requirements are simulated using RSL [91].

Clarke and Lee [94, 95] describe a test generation method that includes timing constraints. Timing constraints are captured graphically through a constraint graph. The Algebra of Communicating Shared Resources (ACSR) is used to represent tests and

process models. Their approach includes testing of timing constraints on the environment (what they call behavioral) as well as constraints on the system (what they call performance). Coverage criteria are given for inputs as well as outputs. Test suites are derived from the constraint graphs. The first step is to derive templates without the timing being specified (i.e., scenarios) by performing a depth first traversal on the constraint graph. Then a second depth first traversal gathers the input delay values that must be covered. All feasible combinations of delays are considered for each template with the optimization directed towards the minimization of the time for test. Finally, a suite of tests is created that provides coverage for their criteria. This is an instance of the set-covering problem so a greedy heuristic is used.

Savor and Saviora [42] report on a method to deal with non-determinism in requirements specifications. The non-determinism is of the type intentionally left in the requirements to give the designer the freedom to do whatever is most cost effective or convenient when any choice will provide the correct behavior. The domain considered is communications systems where most non-determinism is due to message transport delays. The primary contribution is the design of a software supervisor (oracle) that monitors system behavior to detect failures. The system is modeled (SDL is used in the example) and monitored by a hierarchical supervisor with the remainder of the supervisor being written in C++. Their supervisor has a path detection module that looks for unique trajectories in a FSM model in order to determine the path taken by the model. This significantly reduces the computation compared with trying all possible paths. The method for selection of test scenarios is not discussed.

Dalal et al., [46] provide a tool for the automatic generation of test vectors. This paper is a brief overview with references for additional details. Functional requirements are given in terms of system inputs, valid values for the inputs, and constraints on relationships between the inputs (optionally invalid inputs may be specified). The tool will generate a set of test cases that ensures that each value for each input is tested at least once with every other value for every other input. Expected system outputs and timing requirements are not handled.

Ho and Lin [96] discuss a dual language approach that uses temporal logic to formally and precisely describe the system and timed Petri nets as an operational language to describe an abstract model of the behavior of the system to support simulation. The method defines a test segment as a single transition in the Petri net. In essence, all combinations of transitions and all combinations of min/max allowed timing constraints are used to arrive at a set of test cases. The authors acknowledge the time complexity of this test generation approach but do not offer a solution.

En-Nouaary et al., [97, 98] describe how to generate test cases using state characterization techniques for systems specified as Timed Input-Output Automata (TIOA). They define a fault model for such systems that includes timing faults and action/transfer faults. The test architecture assumes that access to internal system state, including timers, is available (i.e., not a black box). The test cases generated by the timed Wp-method are shown to cover all of the faults discussed.

Ammann et al., [99] apply mutation operators and model checking to generate test cases for software. Mutation operators are applied to specifications rather than the more

common target for this technique, which is software. The specification formalism used is SCR but the method is not uniquely tied to it. In addition, temporal logic constraints are needed (which do not contain any explicit references to time so they only impose an ordering of events). The mutations are applied to the specification and to the temporal logic. When a mutated specification and correct temporal logic constraint are fed into a model checker, a counter-example is generated which defines a test that any correct system should fail. When a mutated temporal logic constraint and correct specification are fed into the model checker, a counter-example is generated which defines a test that any correct system should pass. Black box testing concerns are briefly mentioned but no explicit techniques are proposed to help ensure that the tests are useful at this level. Coverage criteria is defined and used to evaluate the generated tests (i.e., the tests were not designed to provide a certain level of coverage – tests are generated and then evaluated by using the coverage criteria as metrics). The authors discovered that their method did not generate tests that cover negative action transitions which are implicit in SCR specifications. They concluded that all of the implicit negative action transitions should be added to the specification and made explicit, a standard practice in many formal methods.

Glover and Cardell-Oliver [100] report on a tool that generates test suites that include timing. Complexity is dealt with by decomposing the model and by varying the temporal granularity. A graph based approach is used where vertices represent state and edges represent timed actions. No details regarding how the tests are generated are given other than stating that different coverage criteria can be selected. In the summary results

given, the number of tests and average test length of the generated tests for the railroad crossing guard appear to be very large.

Gargantini and Heitmeyer [45] describe an approach to automatic test generation proposed by the Naval Research Lab that utilizes model checking. The requirements are specified in their own SCR formalism and a tool developed from previous work that converts the SCR specification to the languages of either the Spin or SMV model checkers is utilized. They then define "trap" properties, which are the negations of the expressions in each state transition of the model. They also defined the coverage criteria they achieved which covers all transitions plus implied negative action transitions. The negative action transitions can either be tested once or once for changes in every monitored variable.

While many of the previous references are closely related to the approach presented in this work, the differences can be summarized into a couple of key areas. The first is the type of system description used as input to the test generation process. While a formal and executable state-based requirements model with separately specified temporal requirements are used here, other approaches use formal logic descriptions (both with and without temporal aspects) [96, 99], finite state machine descriptions (often augmented) [9, 40, 41, 43, 97], timed Petri nets [96], rule-based models [38], input/output specifications [46], the Algebra of Communicating Resources [94], state-based models [45, 99, 100], and more general languages such as the Requirements Specification Language (RSL) [93].

The targeted end use of the generated test suite is also a distinguishing factor. Many researchers target only software implementations, or constrained implementations that allow some visibility into the internal state of the system. Because the target use for this work is to support Model-based Codesign through all levels of the design process, a black box testing model must be used, which complicates the test generation process. The works of Chow [43] and En-Nouaary et. al. [97] address the black box testing issue by applying characterization set sequences to each generated test scenario. This provides a general solution but is costly in terms of the size of the generated test set and En-Nouaary assumes visibility of the internal system clocks. The work of Freeza [93] is applicable to black box testing, but the generated tests are values and not sequences (i.e. only static function and not behavior can be tested). The remaining works cited in this section do not address the black box testing issue.

A third distinguishing factor is the treatment of time. Many of the related works only generate test suites to support the validation of the functional correctness of the system [9, 38, 41, 43, 45, 46, 93, 99]. For real-time systems, temporal correctness is equally important. Temporal correctness is addressed by the tests generated in [40, 94, 96, 97, 100].

8.3 Fault Coverage

A very important aspect of any set of tests is the type of faults detected and to what level of certainty the faults are detected. The fault coverage characteristics of the test scenarios generated as described in the preceding chapters is described in the sub-sections that follow. There are two categories of faults, functional and temporal.

8.3.1 Functional Faults

Functional faults represent the class of faults relating to the input/output relation of the system without consideration of temporal attributes. The general question to be answered is whether the system generates the correct output and only the correct output for each stimulus. This class of faults can be further categorized as described below.

Missing Response Fault: *Required response is not generated by the system.* This type of fault is covered by the generated test scenarios because every state transition changing the value of a system output is covered. However, as mentioned in section 2.1, the type of coverage criteria met by the test set is dependent on the requirements modeler. If a state transition is enabled by multiple stimuli in a disjunctive manner, the generated tests may only test one such input/output relation (i.e., branch cover). In order to assure a complete coverage (i.e., condition cover), disjunctive transitions should not be used in the requirements model.

Extraneous Response Fault: *Response generated by the system for which there is no requirement.* In this case the system generates an unexpected output. Any such output generated by the system during application of one of the test scenarios will be detected by the data analysis algorithm of section 5.4. This is because an exact matching between expected and actual system responses is verified. These types of faults are not targeted during the test scenario generation process. As a result, complete coverage of this type of fault is not guaranteed.

Value Fault: *System response incorrect for given stimulus.* This type of fault occurs when the system generates an response of the proper type, but the value of the response is

not correct. The present test generation algorithms require only that state transitions be exercise and verified. The data values used to exercise the transitions are not considered. As a result, complete coverage of this type of fault is not guaranteed and is dependent on the test input values selected.

Missing State Transition Fault: *The system did not respond to the given stimulus by transitioning to the required state.* As previously discussed, the testing model needed to support Model-based Codesign is that of a black box. As a result, correct state transitions cannot be directly verified. The method presented in sections 2.3.2.3 and 3.3 attempts to generate state identifying sequences (scenario enhancements) by assuming that a faulty system will not include an equivalent to the state transition being verified. Assuming that all state transitions are verifiable by observing the system responses to application of the suite of test scenarios (i.e., scenario enhancements have been successfully generated for all transitions not verified by the base scenarios), then any required state transition not implemented in the system under test will be detected by the generated test scenarios.

Extraneous State Transition Fault: *The system responded to a given stimulus by transitioning to an incorrect state.* Extraneous state transition may not be detected. For example, suppose a system under test properly responds to stimulus A by changing state variable x, but also changes state variable y (a non-required action). Unless the effect of the change to state variable y is observable at a system output during execution of the scenario containing stimulus A, the non-required state change will not be detected. This argument also pertains to scenarios that exercise negative action requirements. These types of transitions are equivalent to self-loops in a state transition graph.

8.3.2 Temporal Faults

Temporal faults represent the class of faults relating to the input/output temporal relation of the system. The general questions to be answered are whether the system accepts stimuli and generates responses within the required temporal bounds. This class of faults can be further categorized as described below.

Late Response Fault: *System response is generated after the maximum allowable time from the triggering event.* The generated test scenarios will exercise all input-output relations and the data analysis algorithm will detect any late system responses. However, due to the one input at a time constraint on the requirements model, parallelism which may impede the ability of a system to meet temporal requirements under heavy input loads (due to resource contention) will not be detected.

Early Response Fault: *System response is generated prior to the minimum allowable time from the triggering event.* Again, the generated test scenarios will exercise all input-output relations and the data analysis algorithm will detect any early system responses. The one input at a time constraint should not diminish the coverage of this type of fault. The reason is that if a system meats the minimum temporal requirements under a one input at a time light load (either by explicit delay or implicitly due the performance of the computing hardware) then additional loading and the possible resource contention could only further delay the system response. In other words, heavy loads may lead to a Late Response Fault but not an Early Response Fault.

Late Stimuli Fault: *System responds to a stimulus applied after the maximum allowable time from the triggering event.* In the proposed use of the C/ATLAS test programs in

section 5.3.3, stimuli will be applied to the system at the boundaries of the temporal requirements for the environment. As a result there is presently no provision to provide a late stimulus to the system. However, it should be a relatively trivial matter to add this capability to the test environment interpreting the C/ATLAS test programs, at the expense of additional test cases.

Early Stimuli Fault: *System responds to a stimulus applied prior to the minimum allowable time from the triggering event.* The dual of the discussion above for the Late Stimuli Fault also applies to Early Stimuli Faults.

8.4 Future Work

As with any work on this scale, time constraints limit the amount that can be accomplished. In addition, observations made during development almost always indicate areas in need of improvement. This section attempts to capture these ideas and suggest topics of future work.

8.4.1 Continued Tool Development

As discussed in chapter 6, several of the algorithms presented in chapter 5 have not been implemented. This includes the synthesis of C/ATLAS test programs, proper interpretation of the C/ATLAS test programs within each test environment, and the analysis of test results. Tools to support these activities are needed to more fully automate the complete test generation, application, and use process as described by Figure 2. These tools should be developed with the considerations discussed in chapter 6.

During the course of developing the Requirements Model Code Synthesis tool, several potential improvements have been noted and recorded and are included in Appendix J. Of particular importance are the items that require the synthesized code to be manually altered before test scenario generation. The single item in this category is that the RMCS tool does not properly synthesize function code for tables with disjunctive expressions that contain both *Inmode* and non-*Inmode* events. An example is from the SISxx models in the Overridden event for requirement ID R6 which is defined to be "@T(Inmode) OR @T(Reset = On)." The correct code corresponding to such an expression is relatively simple but the present architecture of the RMCS tool makes dealing with these expressions difficult. As a result, this has been left as a noted limitation and requires the synthesized code to manually corrected.

8.4.2 Improve Search Heuristics

The heuristics guiding the distance-based and difference-based searches are a likely target for performance improvements. Although the results presented in chapter 7 indicate marginal search efficiency for the distance-based search and reasonable efficiency for the difference-based search, these results may prove to be misleading.

The distance-based search relies on the computation of the distance measures for each potential state relative to the target state. This measure is computed by the ComputeDistance() function synthesized by the RMCS tool. As such, the cost function is embodied in the RMCS tool. The details of the implemented cost function are provided in Appendix K. The presently defined cost function provided adequate results for the final versions of the example systems, but during development of the elevator model

some questionable behavior was observed. Covering certain requirement identifiers proved much more difficult than others. Analysis of one of these cases revealed that one of the potential states available at the start of the search was one input away from being an enabling state for the target requirement identifier. However, it was not selected for expansion because the cost function returned a value of three. Because there were many potential states with cost less than three, search continued until the potential state set contained lowest cost states of three. This suggests that the cost function could be improved by applying a controllability and/or complexity weight associated with the state variables of the model.

The difference-based search used by the scenario enhancement algorithm, *sea*, lacks any heuristic to guide the search. As stated above, the results obtained with the example systems indicate that the difference-based search performed well, but it should be expected to falter for some, probably more complex, systems. Addition of a guiding heuristic has the potential to provide significant performance improvements in such cases. Because the goal of the difference-based search is to propagate a state difference to an observable system output, a cost function that accounted for controllability and distance from a system output would seem a likely candidate.

8.4.3 Improve Fault Coverage

The fault coverage provided by the test programs resulting from the proposed test generation process has been described in section 8.3. The fault classes of *Extraneous Response* and *Extraneous State Transition* faults may not be completely covered by the presently generated suite of test scenarios. These fault classes describe unexpected

behavior from the system under test. Improvements to the coverage of these fault classes would require injecting faults into the requirements model similar to [99]. Another option would be to apply robustness testing which applies inputs based on statistical variation. A negative attribute of this approach is test time and uncertain coverage.

Another class of functional faults for which the generated tests provide poor coverage is for *Value* faults. The input test values are specified prior to scenario generation. The test values are only required to allow the scenario generation algorithms to exercise each state transition but not to exercise the transitions for key data values. This is an area ripe for improvement. The generation algorithms would need to be modified to require multiple coverage of transitions involving comparison of non-enumerated data types. The difficult aspect of this would be determining test input values that would allow the algorithms to fulfill the coverage criteria. This problem could be left in the hands of the modeler, but perhaps a better solution would be to include an adaptive scheme in the algorithm itself that provides automatic experimentation with input values.

The temporal fault classes of *Early* and *Late Stimuli* faults are not presently tested because the criteria as specified in section 5.3.3 is to apply inputs at the inner boundaries of the environmental temporal requirements. This criteria could easily be adjusted to include application of the system inputs at the outer boundaries.

The other class of temporal faults not covered by the generated test suites are the *Late Response* faults. Coverage of this class of faults will require a modeling paradigm that allows the specification of concurrency and the necessary communication

mechanisms to support that concurrency. The scenario generation algorithms, which encapsulate the simulation of the requirements model, would have to be modified to extract the concurrency construct information from the requirements model and then, at least conceptually, provide that concurrency during simulation. The scenario generation algorithms would also need to be altered to implement a coverage criteria that attempts to maximize the concurrency. This would provide a set of test scenarios that would be akin to stress testing and would improve confidence that the system can meet the real-time deadlines under high load conditions. Adding this capability is a difficult problem and has the potential to seriously impact both the test generation time and the size of the resulting suite of test scenarios.

8.4.4 Test generation for Other Classes of Systems

As stated at the outset, the developed scenario generation algorithms have been designed for event-oriented real-time embedded systems. Although, this covers a large class of important embedded systems, there are many embedded systems that do not fit this description and as a result are likely to cause difficulties for the defined algorithms. Such classes of systems include embedded controllers and signal processors. These systems, although usually based on discrete digital systems, process large amounts of continuous valued data. An area of needed research is development of automatic test generation methods for these types of systems.

APPENDIX A SCR MODE L FOR SAFETY INJECTION SYSTEM

```
// This file contains an SCRTool system specification.
// It may have been written by the SCRTool and not changed since.


// This first definition describes the specification as a whole.
// The VERSION is the version of the SCRTool matching this file.
// The DESCRIPTION is the description field for this specification
// as it appears in the main SCRTool window.
SPECIFICATION; VERSION "1.7";


// This section contains all of the items in the
// Type Dictionary.
TYPE "DatalessEventType";     BASETYPE "Enumerated";  UNITS "N/A";
            VALUES "Value1, Value2";
TYPE "SwitchType";       BASETYPE "Enumerated";  UNITS "N/A";
            VALUES "Off, On";


// This section contains all of the items in the
// Mode Class Dictionary.
MODECLASS "Pressure";   MODES "Permitted, TooLow";
            INITMODE "Permitted";


// This section contains all of the items in the
// Constant Dictionary.
CONSTANT "Low";   TYPE "Float";     VAL "100.0";


// This section contains all of the items in the
// Variable Dictionary.
MON "Block";       TYPE "DatalessEventType";     INITVAL "Value1";
MON "Reset";       TYPE "SwitchType";       INITVAL "Off";
CON "SafetyInjection";  TYPE "SwitchType";       INITVAL "Off";
TERM "Overridden";       TYPE "Boolean";   INITVAL "False";
MON "TRef"; TYPE "DatalessEventType";     INITVAL "Value1";
TERM "TRefCnt";   TYPE "Integer";   INITVAL "0";
MON "WaterPres";  TYPE "Integer";   INITVAL "150";


// This section contains all of the items in the
// Specification Assertion Dictionary.


// This section contains all of the items in the
// Environmental Assertion Dictionary.


// This section contains all of the items in the
// Enumerated Monitored Variable Table.
```

```
// This section contains the event, mode transition, and condition
functions.

EVENTFUNC "Overridden"; MCLASS "Pressure";
  MODES "TooLow"  EVENTS "@C(Block) WHEN (Reset=Off)",
      "@T(Inmode)";
  MODES "TooLow"  EVENTS "@T(False)",     "@T(Reset = On)";
  MODES "TooLow"  EVENTS "@T(False)",     "@C(TRef) WHEN (TRefCnt = 2)";
                         ASSIGNMENTS "True",      "False";
  DESCRIPTION "Requirements Identifiers:
R2a   R2b
N/A   R2c
N/A   R4d";

MODETRANS "Pressure";
  FROM "Permitted"       EVENT "@T(WaterPres < Low)"    TO "TooLow";
  FROM "TooLow"    EVENT "@T(WaterPres >= Low)"  TO "Permitted";
  DESCRIPTION "Requirements Identifiers:
P2
P1";

CONDFUNC "SafetyInjection";   MCLASS "Pressure";
  MODES "Permitted"      CONDITIONS "TRUE",       "FALSE";
  MODES "TooLow"  CONDITIONS "Overridden",       "NOT Overridden";
                         ASSIGNMENTS "Off",       "On";
  DESCRIPTION "Requirements Identifiers:
R3a   N/A
R3b   R1";

EVENTFUNC "TRefCnt";     MCLASS "Pressure";
  MODES "TooLow"  EVENTS "@C(TRef) WHEN (Overridden)",
      "@T(false)";
  MODES "TooLow"  EVENTS "@T(false)",    "@C(Block) WHEN (Reset = Off)";
                         ASSIGNMENTS "TRefCnt + 1",     "0";
  DESCRIPTION "Requirements Identifiers:
R4a   N/A
N/A   R4c";
```

APPENDIX B PART 1 ALGORITHM OUTPUT FOR SAFETY INJECTION SYSTEM

```
Scenario Generation part 1                Tue Oct 31 11:26:13 2000

                      Safety Injection System
-------------------------------------------------------------------

Starting Greedy Search with 11 total requirements...
Greedy Search Complete with 2 requirements remaining.
Calling Distance Based Search for requirement R2b.
Calling Distance Based Search for requirement R4d.
SCENARIO GENERATION COMPLETE.
  Number of remaining requirements: 0
  Execution time: 0.010000 seconds

7 UNIQUE SYSTEM STATES HAVE BEEN GENERATED:

  Name   Pressure SafetyInjection     TRefCnt  Overridden
-------------------------------------------------------------
     1  PERMITTED             OFF           0       FALSE
     2  PERMITTED             OFF           0        TRUE
     3    TOOLOW              OFF           0        TRUE
     4    TOOLOW              OFF           1        TRUE
     5    TOOLOW              OFF           2        TRUE
     6    TOOLOW               ON           0       FALSE
     7    TOOLOW               ON           3       FALSE

GENERATED SCENARIOS: [FORMAT: STATE (EQUIV STATE) INPUT / OUTPUTS]

SCENARIO 1
----------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2] [R1]
     6 WaterPres = 150 / SafetyInjection = OFF [P1] [R3a]
     8 (1)

SCENARIO 2
----------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2] [R1]
     6 Block / SafetyInjection = OFF [R2a] [R3b]
     3 WaterPres = 150 / [P1]
     2 WaterPres = 50 / SafetyInjection = ON [P2] [R1] [R2b]
     9 (6)

SCENARIO 3
----------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2] [R1]
     6 Block / SafetyInjection = OFF [R2a] [R3b]
     3 TRef / [R4a]
     4 TRef / [R4a]
     5 TRef / SafetyInjection = ON [R1] [R4a] [R4d]
```

```
     7

SCENARIO 4
----------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2]  [R1]
     6 Block / SafetyInjection = OFF [R2a]  [R3b]
     3 TRef / [R4a]
     4 Block / [R4c]
    10 (3)

SCENARIO 5
----------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2]  [R1]
     6 Block / SafetyInjection = OFF [R2a]  [R3b]
     3 Reset = ON / SafetyInjection = ON [R1]  [R2c]
    11 (6)

ST CONTAINS 11 nodes.

SCENARIO SEARCH TREE (16 total nodes):

Edge # (begin,  end)  Edge Label
------ -------------  ------------------------------------------------
     1 (    1,    6)  'WaterPres = 50 / SafetyInjection = ON'
     2 (    6,    8)  'WaterPres = 150 / SafetyInjection = OFF'
     3 (    6,    3)  'Block / SafetyInjection = OFF'
     4 (    3,    2)  'WaterPres = 150 /'
     5 (    2,    9)  'WaterPres = 50 / SafetyInjection = ON'
     6 (    3,    4)  'TRef /'
     7 (    4,   12)  'WaterPres = 150 /'
     8 (    4,    5)  'TRef /'
     9 (    5,   13)  'WaterPres = 150 /'
    10 (    5,    7)  'TRef / SafetyInjection = ON'
    11 (    5,   14)  'Reset = ON / SafetyInjection = ON'
    12 (    5,   15)  'Block /'
    13 (    4,   16)  'Reset = ON / SafetyInjection = ON'
    14 (    4,   10)  'Block /'
    15 (    3,   11)  'Reset = ON / SafetyInjection = ON'

BINARY SEARCH TREE STATISTICS:
------------------------------
Number of searches of BST required: 20
Minimum number of comparisons needed during a search of BST: 0
Maximum number of comparisons needed during a search of BST: 4
Average number of comparisons needed during a search of BST: 2.45

BINARY SEARCH TREE STATISTICS:
------------------------------
Number of paths (root to pendent) : 3
Minimum path length: 2
Maximum path length: 4
Average path length: 3.00

Total execution time: 0.010000 seconds
```

APPENDIX C PART 2 AL GORITHM OUTPUT FOR SAFETY INJECTION SYSTEM

```
Scenario Generation part 2                Tue Oct 31 11:26:13 2000

                      Safety Injection System
-------------------------------------------------------------------


PROCESSING SCENARIO      1
-------------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2]  [R1]
     6 WaterPres = 150 / SafetyInjection = OFF [P1]  [R3a]
     8

Attempting to verify requirement P2...VERIFIED
Attempting to verify requirement R1...VERIFIED
Attempting to verify requirement P1...VERIFIED
Attempting to verify requirement R3a...VERIFIED

PROCESSING SCENARIO      2
-------------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2]  [R1]
     6 Block / SafetyInjection = OFF [R2a]  [R3b]
     3 WaterPres = 150 / [P1]
     2 WaterPres = 50 / SafetyInjection = ON [P2]  [R1]  [R2b]
     9

Attempting to verify requirement R2a...VERIFIED
Attempting to verify requirement R3b...VERIFIED
Attempting to verify requirement R2b...VERIFIED

PROCESSING SCENARIO      3
-------------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2]  [R1]
     6 Block / SafetyInjection = OFF [R2a]  [R3b]
     3 TRef / [R4a]
     4 TRef / [R4a]
     5 TRef / SafetyInjection = ON [R1]  [R4a]  [R4d]
     7

Attempting to verify requirement R4a...VERIFIED
Attempting to verify requirement R4d...VERIFIED

PROCESSING SCENARIO      4
-------------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2]  [R1]
     6 Block / SafetyInjection = OFF [R2a]  [R3b]
     3 TRef / [R4a]
     4 Block / [R4c]
    10
```

```
Attempting to verify requirement R4c...NOT VERIFIED

PROCESSING SCENARIO      5
--------------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2]  [R1]
     6 Block / SafetyInjection = OFF [R2a]  [R3b]
     3 Reset = ON / SafetyInjection = ON [R1]  [R2c]
    11

Attempting to verify requirement R2c...VERIFIED


Requirements that have not been verified by the given scenarios:
REQ. ID   Earliest occurrence in SCENARIO ID at LEVEL
-------                          -----------    -----
R4c                                        4        4

Total execution time: 0.000000 seconds
```

APPENDIX D PART 3 AL GORITHM OUTPUT FOR SAFETY INJECTION SYSTEM

```
Scenario Generation part 3                Tue Oct 31 11:26:13 2000

                      Safety Injection System
-------------------------------------------------------------------


PROCESSING SCENARIO      4
--------------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2]  [R1]
     6 Block / SafetyInjection = OFF [R2a]  [R3b]
     3 TRef / [R4a]
     4 Block / [R4c]
    10
Enhancing for requirement R4c ... SUCCESSFUL

3 UNIQUE SYSTEM STATES HAVE BEEN GENERATED:

 Name Pressure SafetyInjection TRefCnt Overridden
-------------------------------------------------
     1    TOOLOW               OFF           0        TRUE
     3    TOOLOW               OFF           1        TRUE
     7    TOOLOW               OFF           2        TRUE

Enhancement for scenario 4:
----------------------------
     1 TRef / [R4a]
     3 TRef / [R4a]
     7

ST CONTAINS 3 nodes.

SCENARIO SEARCH TREE (7 total nodes):

Edge # (begin,  end)  Edge Label
------ ------------  ------------------------------------------------
     1 (    1,    2)  'WaterPres = 150 /'
     2 (    2,    5)  'WaterPres = 50 / SafetyInjection = ON'
     3 (    1,    3)  'TRef /'
     4 (    3,    6)  'WaterPres = 150 /'
     5 (    3,    7)  'TRef /'
     6 (    1,    4)  'Reset = ON / SafetyInjection = ON'

System output differences detected:
-----------------------------------
SafetyInjection:  Good Value = OFF  Bad Value = ON

Total execution time: 0.000000 seconds
```

APPENDIX E PART 4 AL GORITHM OUTPUT FOR SAFETY INJECTION SYSTEM

```
Scenario Generation part 4                Tue Oct 31 11:26:13 2000

                      Safety Injection System
-------------------------------------------------------------------

Processing base states from SGA output file sis1.out...Complete
Processing base scenarios from SGA output file sis3.out...Complete
Processing scenario enhancements from SE output file
sis3.out...Complete

7 UNIQUE SYSTEM STATES HAVE BEEN GENERATED:

  Name    Pressure SafetyInjection     TRefCnt  Overridden
-------------------------------------------------------------
     1  PERMITTED             OFF           0       FALSE
    8 = 1
     2  PERMITTED             OFF           0        TRUE
     3    TOOLOW              OFF           0        TRUE
   10 = 3
     4    TOOLOW              OFF           1        TRUE
   12 = 4
     5    TOOLOW              OFF           2        TRUE
   13 = 5
     6    TOOLOW               ON           0       FALSE
    9 = 6
   11 = 6
     7    TOOLOW               ON           3       FALSE


GENERATED SCENARIOS: [FORMAT: STATE INPUT / OUTPUTS]

SCENARIO 1
----------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2] [R1]
     6 WaterPres = 150 / SafetyInjection = OFF [P1] [R3a]
     8

SCENARIO 2
----------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2] [R1]
     6 Block / SafetyInjection = OFF [R2a] [R3b]
     3 WaterPres = 150 / [P1]
     2 WaterPres = 50 / SafetyInjection = ON [P2] [R1] [R2b]
     9

SCENARIO 3
----------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2] [R1]
```

```
     6 Block / SafetyInjection = OFF [R2a]  [R3b]
     3 TRef / [R4a]
     4 TRef / [R4a]
     5 TRef / SafetyInjection = ON [R1]  [R4a]  [R4d]
     7

SCENARIO 4
----------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2]  [R1]
     6 Block / SafetyInjection = OFF [R2a]  [R3b]
     3 TRef / [R4a]
     4 Block / [R4c]
    10 TRef / [R4a]
    12 TRef / [R4a]
    13

SCENARIO 5
----------------------
     1 WaterPres = 50 / SafetyInjection = ON [P2]  [R1]
     6 Block / SafetyInjection = OFF [R2a]  [R3b]
     3 Reset = ON / SafetyInjection = ON [R1]  [R2c]
    11

ST CONTAINS 13 nodes.

SCENARIO SEARCH TREE (13 total nodes):

Edge# (begin,end) Edge Label
----- ----------- ------------------------------------------------
    1 (    1,  6) 'WaterPres = 50 / SafetyInjection = ON [P2][R1]'
    2 (    6,  8) 'WaterPres = 150 / SafetyInjection = OFF [P1][R3a]'
    3 (    6,  3) 'Block / SafetyInjection = OFF [R2a][R3b]'
    4 (    3,  2) 'WaterPres = 150 / [P1]'
    5 (    2,  9) 'WaterPres = 50 / SafetyInjection = ON [P2][R1][R2b]'
    6 (    3,  4) 'TRef / [R4a]'
    7 (    4,  5) 'TRef / [R4a]'
    8 (    5,  7) 'TRef / SafetyInjection = ON [R1][R4a][R4d]'
    9 (    4, 10) 'Block / [R4c]'
   10 (   10, 12) 'TRef / [R4a]'
   11 (   12, 13) 'TRef / [R4a]'
   12 (    3, 11) 'Reset = ON / SafetyInjection = ON [R1][R2c]'

BINARY SEARCH TREE STATISTICS:
------------------------------
Number of paths (root to pendent) : 4
Minimum path length: 2
Maximum path length: 2
Average path length: 2.00

Total execution time: 0.000000 seconds
```

APPENDIX F C/ATLAS TEST PROGRAMS FOR SIS TEST SCENARIOS

```
C       ***************************************************$
C       C/ATLAS TEST PROGRAM FOR SAFETY INJECTION SYSTEM $
C       ***************************************************$
C       $
 001000 BEGIN, ATLAS PROGRAM 'SAFETY INJECTION SYSTEM TEST 1' $
C       $
 001001 EXTEND, ATLAS NOUN, 'ABSTRACT_SIGNAL',
         FOR VERBS APPLY,
         SPEC 'NONE' $
 001002 EXTEND, ATLAS MODIFIERS FOR ABSTRACT_SIGNAL,
         'SIGNAL-TYPE',
            MOD-TYPE MNEMONIC-ONLY,
            USAGE STIMULUS-RESPONSE,
         'LEGAL-VALUES',
           USAGE STIMULUS-RESPONSE,
         'PRESENT-VALUE',
           USAGE STIMULUS $
 001003 EXTEND, ATLAS NOUN, 'DATALESS_EVENT' $
C       $
C       Define signals $
C       $
 002000 DEFINE, 'WATERPRESSURE', SIGNAL, ABSTRACT_SIGNAL,
         SIGNAL-TYPE REAL, LEGAL-VALUES RANGE 0.0 TO 300.0,
         PRESENT-VALUE () $
 002001 DEFINE, 'BLOCK', SIGNAL, ABSTRACT_SIGNAL,
         SIGNAL-TYPE DATALESS_EVENT $
 002002 DEFINE, 'RESET', SIGNAL, ABSTRACT_SIGNAL,
         SIGNAL-TYPE ENUMERATED, LEGAL-VALUES {'ON', 'OFF'},
         PRESENT-VALUE () $
 002003 DEFINE, 'TREF', SIGNAL, ABSTRACT_SIGNAL,
         SIGNAL-TYPE DATALESS_EVENT $
 002004 DEFINE, 'SAFETYINJECTION', SIGNAL, ABSTRACT_SIGNAL,
         SIGNAL-TYPE ENUMERATED, LEGAL-VALUES {'ON', 'OFF'} $
C       $
C       Identify all needed events $
C       $
 003000 IDENTIFY, EVENT 'SI_OFF' AS 'SAFETYINJECTION' EQ 'OFF' $
 003001 IDENTIFY, EVENT 'SI_ON' AS 'SAFETYINJECTION' EQ 'ON' $
C       $
C       Set up initial input values $
C       $
 004000 APPLY, 'WATERPRESSURE', 150.0 $
 004001 APPLY, 'RESET', 'OFF', $
C       $
C       Enable monitoring of events $
C       $
 005000 ENABLE, EVENT 'SI_ON', 'SI_OFF' $
C       $
C       Start the test scenario $
C       $
 006000 WAIT FOR, 1 SEC $
```

```
 006001 APPLY, 'WATERPRESSURE', 50.0 $
 006002 WAIT FOR, EVENT 'SI_ON', MAX-TIME 1 SEC $
 006003 APPLY, 'WATERPRESSURE', 150.0 $
 006004 WAIT FOR, EVENT 'SI_OFF', MAX-TIME 500 MSEC $
C      $
 999999 TERMINATE, ATLAS PROGRAM 'SAFETY INJECTION SYSTEM 1' $


C      ************************************************$
C      C/ATLAS TEST PROGRAM FOR SAFETY INJECTION SYSTEM $
C      ************************************************$
C      $
 001000 BEGIN, ATLAS PROGRAM 'SAFETY INJECTION SYSTEM TEST 2' $
C      $
 001001 EXTEND, ATLAS NOUNS, 'ABSTRACT_SIGNAL',
        FOR VERBS APPLY,
        SPEC 'NONE' $
 001002 EXTEND, ATLAS MODIFIERS FOR ABSTRACT_SIGNAL,
        'SIGNAL-TYPE',
           MOD-TYPE MNEMONIC-ONLY,
           USAGE STIMULUS-RESPONSE,
        'LEGAL-VALUES',
          USAGE STIMULUS-RESPONSE,
        'PRESENT-VALUE',
          USAGE STIMULUS $
 001003 EXTEND, ATLAS NOUNS, 'DATALESS_EVENT' $
C      $
C      Define signals $
C      $
 002000 DEFINE, 'WATERPRESSURE', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE REAL, LEGAL-VALUES RANGE 0.0 TO 300.0,
        PREESENT-VALUE () $
 002001 DEFINE, 'BLOCK', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE DATALESS_EVENT $
 002002 DEFINE, 'RESET', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE ENUMERATED, LEGAL_VALUES {'ON', 'OFF'},
        PRESENT-VALUE () $
 002003 DEFINE, 'TREF', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE DATALESS_EVENT $
 002004 DEFINE, 'SAFETYINJECTION', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE ENUMERATED, LEGAL-VALUES {'ON', 'OFF'} $
C      $
C      Identify all needed events $
C      $
 003000 IDENTIFY, EVENT 'SI_ON' AS 'SAFETYINJECTION' EQ 'ON' $
 003001 IDENTIFY, EVENT 'SI_OFF' AS 'SAFETYINJECTION' EQ 'OFF' $
C      $
C      Set up initial input values $
C      $
 004000 APPLY, 'WATERPRESSURE', 150.0 $
 004001 APPLY, 'RESET', 'OFF', $
C      $
C      Enable monitoring of events $
C      $
```

```
 005000 ENABLE, EVENT 'SI_ON', 'SI_OFF' $
C      $
C      Start the test scenario $
C      $
 006000 WAIT FOR, 1 SEC $
 006001 APPLY, 'WATERPRESSURE', 50.0 $
 006002 WAIT FOR, EVENT 'SI_ON', MAX-TIME 1 SEC $
 006003 APPLY, 'BLOCK' $
 006004 WAIT FOR, EVENT 'SI_OFF', MAX-TIME 1.5 SEC $
 006005 APPLY, 'WATERPRESSURE', 150.0 $
 006006 APPLY, 'WATERPRESSURE', 50.0 $
 006007 WAIT FOR, EVENT 'SI_ON', MAX-TIME 1 SEC $
C      $
 999999 TERMINATE, ATLAS PROGRAM 'SAFETY INJECTION SYSTEM 1' $


C      ************************************************$
C      C/ATLAS TEST PROGRAM FOR SAFETY INJECTION SYSTEM $
C      ************************************************$
C      $
 001000 BEGIN, ATLAS PROGRAM 'SAFETY INJECTION SYSTEM TEST 3' $
C      $
 001001 EXTEND, ATLAS NOUNS, 'ABSTRACT_SIGNAL',
        FOR VERBS APPLY $
 001002 EXTEND, ATLAS MODIFIERS FOR ABSTRACT_SIGNAL,
        'SIGNAL-TYPE',
           MOD-TYPE MNEMONIC-ONLY,
           USAGE STIMULUS-RESPONSE,
        'LEGAL-VALUES',
          USAGE STIMULUS-RESPONSE,
        'PRESENT-VALUE',
          USAGE STIMULUS $
 001003 EXTEND, ATLAS NOUNS, 'DATALESS_EVENT' $
C      $
C      Define signals $
C      $
 002000 DEFINE, 'WATERPRESSURE', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE REAL, LEGAL-VALUES RANGE 0.0 TO 300.0,
        PRESENT-VALUE () $
 002001 DEFINE, 'BLOCK', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE DATALESS_EVENT $
 002002 DEFINE, 'RESET', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE ENUMERATED, LEGAL_VALUES {'ON', 'OFF'},
        PRESENT-VALUE () $
 002003 DEFINE, 'TREF', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE DATALESS_EVENT $
 002004 DEFINE, 'SAFETYINJECTION', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE ENUMERATED, LEGAL-VALUES {'ON', 'OFF'} $
C      $
C      Identify all needed events $
C      $
 003000 IDENTIFY, EVENT 'SI_ON' AS 'SAFETYINJECTION' EQ 'ON' $
 003001 IDENTIFY, EVENT 'SI_OFF' AS 'SAFETYINJECTION' EQ 'OFF' $
C      $
```

```
C       Set up initial input values $
C       $
 004000 APPLY, 'WATERPRESSURE', 150.0 $
 004002 APPLY, 'RESET', 'OFF', $
C       $
C       Enable monitoring of events $
C       $
 005000 ENABLE, EVENT 'SI_ON', 'SI_OFF' $
C       $
C       Start the test scenario $
C       $
 006000 WAIT FOR, 1 SEC $
 006001 APPLY, 'WATERPRESSURE', 50.0 $
 006002 WAIT FOR, EVENT 'SI_ON', MAX-TIME 1 SEC $
 006003 APPLY, 'BLOCK' $
 006004 WAIT FOR, EVENT 'SI_OFF', MAX-TIME 1.5 SEC $
 006005 APPLY, 'TREF' $
 006006 APPLY, 'TREF' $
 006007 APPLY, 'TREF' $
 006008 WAIT FOR, EVENT 'SI_ON', MAX-TIME 3 SEC $
C       $
 999999 TERMINATE, ATLAS PROGRAM 'SAFETY INJECTION SYSTEM 1' $


C       ************************************************$
C       C/ATLAS TEST PROGRAM FOR SAFETY INJECTION SYSTEM $
C       ************************************************$
C       $
 001000 BEGIN, ATLAS PROGRAM 'SAFETY INJECTION SYSTEM TEST 4' $
C       $
 001001 EXTEND, ATLAS NOUNS, 'ABSTRACT_SIGNAL',
         FOR VERBS APPLY $
 001002 EXTEND, ATLAS MODIFIERS FOR ABSTRACT_SIGNAL,
         'SIGNAL-TYPE',
            MOD-TYPE MNEMONIC-ONLY,
            USAGE STIMULUS-RESPONSE,
         'LEGAL-VALUES',
           USAGE STIMULUS-RESPONSE,
         'PRESENT-VALUE',
           USAGE STIMULUS $
 001003 EXTEND, ATLAS NOUNS, 'DATALESS_EVENT' $
C       $
C       Define signals $
C       $
 002000 DEFINE, 'WATERPRESSURE', SIGNAL, ABSTRACT_SIGNAL,
         SIGNAL-TYPE REAL, LEGAL-VALUES RANGE 0.0 TO 300.0,
         PRESENT-VALUE () $
 002001 DEFINE, 'BLOCK', SIGNAL, ABSTRACT_SIGNAL,
         SIGNAL-TYPE DATALESS_EVENT $
 002002 DEFINE, 'RESET', SIGNAL, ABSTRACT_SIGNAL,
         SIGNAL-TYPE ENUMERATED, LEGAL_VALUES {'ON', 'OFF'},
         PRESENT-VALUE () $
 002003 DEFINE, 'TREF', SIGNAL, ABSTRACT_SIGNAL,
         SIGNAL-TYPE DATALESS_EVENT $
```

```
 002004 DEFINE, 'SAFETYINJECTION', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE ENUMERATED, LEGAL-VALUES {'ON', 'OFF'} $
C       $
C       Identify all needed events $
C       $
 003000 IDENTIFY, EVENT 'SI_ON' AS 'SAFETYINJECTION' EQ 'ON' $
 003001 IDENTIFY, EVENT 'SI_OFF' AS 'SAFETYINJECTION' EQ 'OFF' $
C       $
C       Set up initial input values $
C       $
 004000 APPLY, 'WATERPRESSURE', 150.0 $
 004002 APPLY, 'RESET', 'OFF', $
C       $
C       Enable monitoring of events $
C       $
 005000 ENABLE, EVENT 'SI_ON', 'SI_OFF' $
C       $
C       Start the test scenario $
C       $
 006000 WAIT FOR, 1 SEC $
 006001 APPLY, 'WATERPRESSURE', 50.0 $
 006002 WAIT FOR, EVENT 'SI_ON', MAX-TIME 1 SEC $
 006003 APPLY, 'BLOCK' $
 006004 WAIT FOR, EVENT 'SI_OFF', MAX-TIME 1.5 SEC $
 006005 APPLY, 'TREF' $
 006006 APPLY, 'BLOCK' $
 006007 APPLY, 'TREF' $
 006008 APPLY, 'TREF' $
C       $
 999999 TERMINATE, ATLAS PROGRAM 'SAFETY INJECTION SYSTEM 1' $


C       ************************************************$
C       C/ATLAS TEST PROGRAM FOR SAFETY INJECTION SYSTEM $
C       ************************************************$
C       $
 001000 BEGIN, ATLAS PROGRAM 'SAFETY INJECTION SYSTEM TEST 5' $
C       $
 001001 EXTEND, ATLAS NOUNS, 'ABSTRACT_SIGNAL',
        FOR VERBS APPLY $
 001002 EXTEND, ATLAS MODIFIERS FOR ABSTRACT_SIGNAL,
        'SIGNAL-TYPE',
           MOD-TYPE MNEMONIC-ONLY,
           USAGE STIMULUS-RESPONSE,
        'LEGAL-VALUES',
          USAGE STIMULUS-RESPONSE,
        'PRESENT-VALUE',
          USAGE STIMULUS $
 001003 EXTEND, ATLAS NOUNS, 'DATALESS_EVENT' $
C       $
C       Define signals $
C       $
 002000 DEFINE, 'WATERPRESSURE', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE REAL, LEGAL-VALUES RANGE 0.0 TO 300.0,
```

```
        PRESENT-VALUE () $
 002001 DEFINE, 'BLOCK', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE DATALESS_EVENT $
 002002 DEFINE, 'RESET', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE ENUMERATED, LEGAL_VALUES {'ON', 'OFF'},
        PRESENT-VALUE () $
 002003 DEFINE, 'TREF', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE DATALESS_EVENT $
 002004 DEFINE, 'SAFETYINJECTION', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE ENUMERATED, LEGAL-VALUES {'ON', 'OFF'} $
C       $
C       Identify all needed events $
C       $
 003000 IDENTIFY, EVENT 'SI_ON' AS 'SAFETYINJECTION' EQ 'ON' $
 003001 IDENTIFY, EVENT 'SI_OFF' AS 'SAFETYINJECTION' EQ 'OFF' $
C       $
C       Set up initial input values $
C       $
 004000 APPLY, 'WATERPRESSURE', 150.0 $
 004002 APPLY, 'RESET', 'OFF', $
C       $
C       Enable monitoring of events $
C       $
 005000 ENABLE, EVENT 'SI_ON', 'SI_OFF' $
C       $
C       Start the test scenario $
C       $
 006000 WAIT FOR, 1 SEC $
 006001 APPLY, 'WATERPRESSURE', 50.0 $
 006002 WAIT FOR, EVENT 'SI_ON', MAX-TIME 1 SEC $
 006003 APPLY, 'BLOCK' $
 006004 WAIT FOR, EVENT 'SI_OFF', MAX-TIME 1.5 SEC $
 006005 APPLY, 'RESET', On $
 006006 WAIT FOR, EVENT 'SI_ON', MAX-TIME 2 SEC $
C       $
 999999 TERMINATE, ATLAS PROGRAM 'SAFETY INJECTION SYSTEM 1' $
```

APPENDIX G EXPRESSIO N GRAMMAR FOR RMCS TOOL

expression ::=	full_ifandonlyif | o_event
full_ifandonlyif ::=	full_implication ifandonlyif
ifandonlyif ::=	IFF full_implication ifandonlyif | ε
full_implication ::=	o_cond implication
implication ::=	IMPLIES o_cond implication | ε
o_event ::=	a_event | o_event OR a_event | o_event OR full_ifandonlyif
a_event ::=	simple_event | a_event AND simple_event | a_event AND full_ifandonlyif
simple_event ::=	event | cond_event | not_event | LPAREN o_event close_expression | ALWAYS | NEVER
o_cond ::=	a_cond | o_cond OR a_cond
a_cond ::=	simple_cond | a_cond AND simple_cond
simple_cont ::=	not_cond | relat_exp | inmode_exp | arb_exp
not_cond ::=	NOT literal | NOT variable | NOT inmode_exp | NOT LPAREN full_ifandonlyif close_expression | NOT LPAREN not_cond close_expression
not_event ::=	NOT event | NOT cond_event | NOT LPAREN not_event close_expression | NOT LPAREN event close_expression | NOT LPAREN cond_event close_expression
relat_exp ::=	arb_exp RELOP arb_exp
arb_exp ::=	arith_exp | u_arith_exp | literal | LPAREN full_ifandonlyif close_expression
arith_exp ::=	term | arith_exp MINUS term | arith_exp PLUS term
u_arith_exp ::=	u_term | u_arith_exp MINUS term | u_arith_exp PLUS term

term ::=	num_literal | variable | term MULT num_literal | term DIV num_literal | term MULT variable | term DIV variable
u_term ::=	unary_term | u_term MULT num_literal | u_term DIV num_literal | u_term MULT variable | u_term DIV variable
unary_term ::=	MINUS num_literal | MINUS variable | PLUS num_literal | PLUS variable
cond_event ::=	event WHEN full_ifandonlyif
event ::=	AT_TRUE LPAREN full_ifandonlyif close_expression | AT_FALSE LPAREN full_ifandonlyif close_expression | VALUE_CHANGE LPAREN full_ifandonlyif close_expression | STATE_CHANGE LPAREN full_ifandonlyif close_expression
inmode_exp ::=	INMODE
literal ::=	CHARACTER | BOOL_TRUE | BOOL_FALSE
num_literal ::=	INTEGER | FLOAT
variable ::=	NAME | NAME PRIME | LPAREN arith_exp close_expression | LPAREN u_arith_exp close_expression
close_expression ::=	RPAREN | RPAREN PRIME
RPAREN ::=	")"
PRIME ::=	"'"
LPAREN ::=	"("
INTEGER ::=	[0-9]+
FLOAT ::=	[0-9]*"."[0-9]+
NAME ::=	[A-Za-z]+[A-Za-z0-9_]*
CHARACTER ::=	[A-Za-z0-9!@#$%^&*()_+|~`\\=-[\]{};':",./<>?]
BOOL_TRUE ::=	[Tt][Rr][Uu][Ee]
BOOL_FALSE ::=	[Ff][Aa][Ll][Ss][Ee]
INMODE ::=	[Ii][Nn][Mm][Oo][Dd][Ee]
AT_TRUE ::=	"@T"
AT_FALSE ::=	"@F"
VALUE_CHANGE ::=	"@C"
STATE_CHANGE ::=	"@A"
WHEN ::=	[Ww][Hh][Ee][Nn]
MINUS ::=	"-"
PLUS ::=	"+"
MULT ::=	"*"
DIV ::=	"/"

RELOP ::=	"=" | "!=" | "<" | "<=" | ">" | ">="
NOT ::=	[Nn][Oo][Tt]
AND ::=	[Aa][Nn][Dd]
OR ::=	[Oo][Rr]
ALWAYS ::=	[Aa][Ll][Ww][Aa][Yy][Ss]
NEVER ::=	[Nn][Ee][Vv][Ee][Rr]
IMPLIES ::=	"=>"
IFF ::=	"<=>"

APPENDIX H REQUIREMENTS FOR THE ELEVATOR CONTROLLER

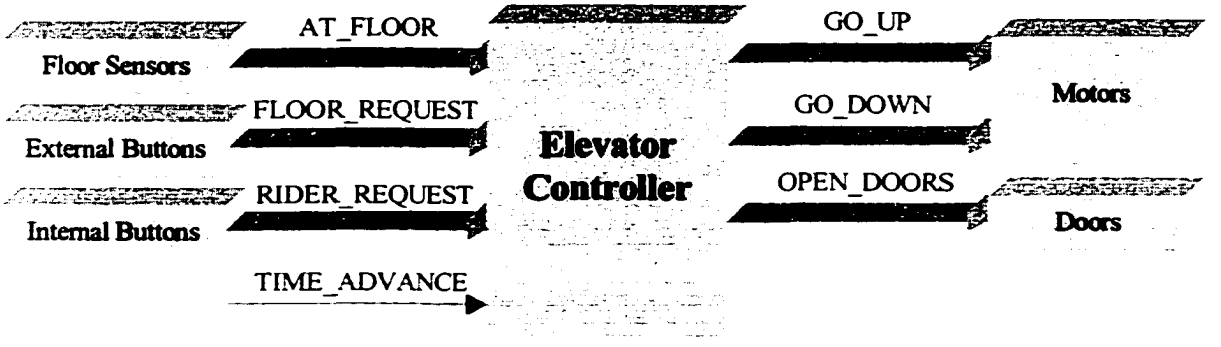


Figure 41. Elevator Controller I/O Block Diagram

[R1] When the elevator is not presently servicing any requests, the elevator shall respond to an external request by asserting GO_UP or GO_DOWN as appropriate to start moving toward the floor from which the request was made within 0.5 seconds of when the request was made.

[R2] If the elevator is in the process of servicing a request, and a new request is received (either internal or external) which can be serviced without changing direction, that request will be serviced prior to completing the present request.

[R3] A service request is completed when the elevator has arrived at the proper floor and the doors have been open for 5 seconds.

[R4] When the elevator arrives at a floor in response to a request, the doors shall be commanded to open by asserting OPEN_DOORS within 0.5 seconds.

[R5] When a service request is complete, and other requests are pending, the next request to be serviced shall be the request at the closest floor in the same direction as last motion, or if there are no requests in the same direction, the request at the closest floor in the opposite direction of the last motion shall be selected.

[R6] The doors shall be commanded to close by de-asserting OPEN_DOORS within 0.5 seconds of completing a service request.

[R7] The elevator shall be commanded to move by asserting GO_UP or GO_DOWN, if there are pending service requests, within 0.5 seconds of the doors becoming closed.

[R8] In the absence of any service requests, the elevator shall remain at the present floor.

APPENDIX I TEST GENERATION MEASURED DATA

Table 17. Coverage by Algorithm

Model	Covered by Greedy Search	Covered by Distance-based Search	Total
SIS	9	2	11
SISsm	2	8	10
SISIg	2	8	10
SISsmNA	16	8	24
SISIgNA	9	15	24
Temperature Controller	31	0	31
Elevator Controller	51	12	63

Table 18. Black Box Verifiable Requirements

Model	Verifiable with Base Scenarios	Verifiable with Scenario Enhancements	Unverifiable at black-box Level
SIS	10	1	0
SISsm	6	3	1
SISIg	6	3	1
SISsmNA	6	3	15
SISIgNA	6	3	15
Temperature Controller	24	5	2
Elevator Controller	58	5	0

Table 19. Length of Generated Scenarios (restricted *WaterPres* changes)

Model	Number of Scenarios	Average Scenario Length	Total Length All Scenarios	Individual Scenario Lengths
SIS	5	4	20	2, 4, 5, 6, 3
SISsm	6	4.33	26	6, 7, 3, 3, 3, 4
SISlg	6	10.17	61	16, 17, 6, 7, 7, 8
SISsmNA	12	3.58	43	3, 6, 3, 7, 3, 3, 3, 3, 3, 3, 4, 2
SISlgNA	12	7.67	92	6, 16, 6, 17, 7, 7, 7, 7, 6, 3, 8, 2
Temperature Controller	12	4.67	56	1, 2, 6, 7, 7, 6, 6, 7, 7, 5, 1, 1
Elevator Controller	22	7.95	176	6, 8, 8, 12, 16, 14, 12, 12, 16, 18, 16, 7, 7, 7, 1, 1, 5, 3, 3, 2, 1, 1

Table 20. Length of Generated Scenarios (unrestricted *WaterPres* changes)

Model	Number of Scenarios	Average Scenario Length	Total Length All Scenarios	Individual Scenario Lengths
SIS	5	4.00	20	2, 4, 5, 6, 3
SISsm	5	2.80	14	3, 4, 2, 3, 2
SISlg	5	2.80	14	3, 4, 2, 3, 2
SISsmNA	12	3.58	43	3, 6, 3, 7, 3, 3, 3, 3, 3, 3, 4, 2
SISlgNA	12	3.58	43	3, 6, 3, 7, 3, 3, 3, 3, 3, 3, 4, 2
Temperature Controller	12	4.67	56	1, 2, 6, 7, 7, 6, 6, 7, 7, 5, 1, 1
Elevator Controller	22	8.00	176	6, 8, 8, 12, 16, 14, 12, 12, 16, 18, 16, 7, 7, 7, 1, 1, 5, 3, 3, 2, 1, 1

Table 21. Search Efficiency: ST Vs. SST

	Base Scenarios			Scenario Enhancements				
Model	States in ST	States in SST	Percent Productive	States in ST	Total	States in SST	Total	Percent Productive
SIS	11	16	68.75%	3	3	7	7	42.86%
SISsm	9	12	75.00%	6, 4, 3	13	28, 10, 6	44	29.55%
SISIg	9	15	60.00%	13, 11, 7	31	89, 52, 6	147	21.09%
SISsmNA	19	41	46.34%	6, 4, 3	13	52, 16, 10	78	16.67%
SISIgNA	19	41	46.34%	6, 4, 3	13	59, 16, 10	78	16.67%
Temperature Controller	21	143	14.69%	3, 3, 2, 3, 3	14	11, 12, 4, 11, 12	50	28.00%
Elevator Controller	61	275	22.18%	4, 2, 2, 2, 2	12	26, 3, 4, 5, 5	43	27.91%

Table 22. Test Suite Efficiency

Model	Test Suite Efficiency	Overlap per Scenario	Total Overlap	Compensated Test Suite Efficiency
SIS	0.55	0, 1, 2, 3, 2	8	0.92
SISsm	0.71	0, 1, 0, 1, 0	2	0.83
SISIg	0.71	0, 1, 0, 1, 0	2	0.83
SISsmNA	0.56	0, 2, 2, 2, 0, 2, 2, 2, 0, 1, 2, 1	16	0.89
SISIgNA	0.56	0, 2, 2, 2, 0, 2, 2, 2, 0, 1, 2, 1	16	0.89
Temperature Controller	0.55	0, 0, 1, 4, 4, 4, 3, 4, 4, 4, 0, 0	28	1.11
Elevator Controller	0.36	0, 3, 2, 6, 9, 13, 10, 10, 10, 12, 14, 6, 6, 5, 0, 0, 0, 2, 0, 1, 0, 0	109	0.94

Table 23. Execution Times

Model	Part 1 (sga)	Part 2 (sva)	Part 3 (sea)	Part 4 (sca)	Total
SIS	0.01	0.00	0.00	0.00	0.01
SISsm	0.00	0.00	0.01	0.01	0.02
SISIg	0.01	0.01	0.01	0.01	0.04
SISsmNA	0.01	0.02	0.03	0.02	0.08
SISIgNA	0.01	0.01	0.08	0.02	0.12
Temperature Controller	0.03	0.02	0.03	0.01	0.09
Elevator Controller	0.11	0.15	0.06	0.06	0.38

Table 24. Measured Binary Search Tree Data

Model	Number of Paths	Minimum Path Length	Maximum Path Length	Average Path Length	Number of Vertices
SIS	3	2	4	3.00	7
SISsm	2	3	4	3.50	8
SISsmNA	5	3	4	3.40	13
Temperature Controller	6	1	7	5.00	19
SISIg	9	2	7	4.78	22
SISIgNA	11	3	10	5.82	32
Elevator Controller	25	3	12	8.04	69

Table 25. Optimal Binary Search Tree Data

Model	Optimal Number of Paths	Optimal Minimum Path Length	Optimal Maximum Path Length	Optimal Average Path Length	Number of Vertices
SIS	4	2	2	2.00	7
SISsm	4	2	3	2.25	8
SISsmNA	7	2	3	2.86	13
Temperature Controller	10	3	4	3.40	19
SISIg	11	3	4	3.64	22
SISIgNA	16	4	5	4.06	32
Elevator Controller	35	5	6	5.17	69

APPENDIX J RMCS TOOL PLANNED IMPROVEMENTS

1. A more complete definition of the grammar used in SCR is needed. Some of the valid constructs are not outlined in the draft User Manual.

2. The tool could be made more robust in terms of portability (e.g, by using JAVA), functionality (e.g., by providing a GUI), error checking, error handling, etc. Also since the Naval Research Labs have already developed a JAVA API as part of SCRTool2, a class of libraries already exists that could be utilized to support this effort.

3. Rather than the tool terminating execution upon encountering an error, it should generate as much output as it can and then produce a summary of all functions synthesized and those that were not synthesized, and reasons why they were not.

4. The SCR Tool set doesn't seem to perform type checking in the function tables (e.g., assignment of a literal float value to a variable of type integer does not raise an error). A more robust method is needed to synthesize the ComputeDistance() function if constraints on variable change is to be taken into account.

5. Tool should be improved to handle multiple delta values for the inputs and to enforce an ordering for input values specified as absolute.

6. Provide a method for this tool to automatically synthesize code that causes the test generation algorithms to generate tests for negative action transitions.

7. A present limitation is that the current Modes listed in modeclass tables must be grouped in adjacent rows. The tool should be enhanced to handle random orders of rows in modeclass tables.

8. A present limitation is that the tool cannot properly synthesize function code for event tables containing both an Inmode event and other types of events. The tool should be improved to handle these instances.

9. Improve the presentation of output defined in output functions (e.g., alignment in PrintState()).

10. Need a better way to handle precision of variables as defined in output functions (how many digits of precision are required for integers and floats).

APPENDIX K IMPLEMENTED COMPUTEDISTANCE() FUNCTION

Expressions, either event or condition, involving enumerated types (includes logical or boolean types) are given a distance of 1 or 0 depending on the context. Table 26 below defines the distance calculations associated with expressions involving numeric types.

Table 26. ComputeDist() Cost Function for Numeric Types

SCR Context	Case	Expression	Distance Calculation

@T(expression) or an expression involving a primed variable	1	a < b
		a' < b
		a < b'
			distance += fabs(a - b);
			if (a < b)
			  distance += 10.0;

	2	a > b
		a' > b
		a > b'
			distance += fabs(a - b);
			if (a > b)
			  distance += 10.0;

	3	a <= b
		a' <= b
		a <= b'
			distance += fabs(a - (b + 1));
			if (a <= b)
			  distance += 10.0;

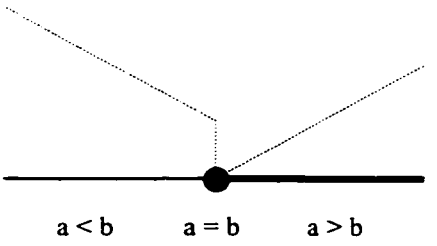
	4	a >= b
		a' >= b
		a >= b'
			distance += fabs(a - (b - 1));
			if (a >= b)
			  distance += 10.0;

	5	a = b
		a' = b
		a = b'
			distance += fabs(a - b)
			if (a = b)
			  distance += 10.0;

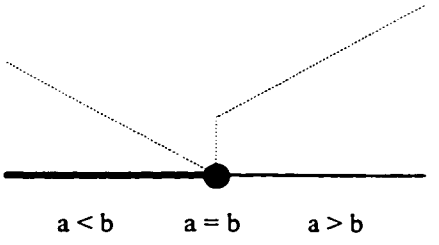
	6	a != b a' != b a != b'	if (a != b) distance += 1.0;
@F(expression)	1		Same as case 4 from @T() above.
	2		Same as case 3 from @T() above.
	3		Same as case 2 from @T() above.
	4		Same as case 1 from @T() above.
	5		Same as case 6 from @T() above.
	6		Same as case 5 from @T() above.
WHEN expression	7	a < b	if (a >= b) distance += 1.0;
	8	a > b	if (a <= b) distance += 1.0;
	9	a <= b	if (a > b) distance += 1.0;
	10	a >= b	if (a < b) distance += 1.0;
	11	a = b	distance += fabs(a - b);
	12	a != b	if (a = b) distance += 1.0;

The distance functions for the 12 cases listed in Table 26 are illustrated below. In these figures the horizontal axis represents the relationship between a and b and the vertical axis (not shown) is the distance. A bold line indicates an acceptable relationship to enable the needed state transition. The dashed line represents the distance function.

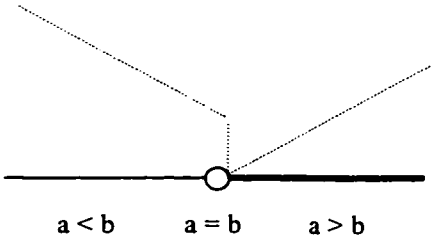
Case 1: .....



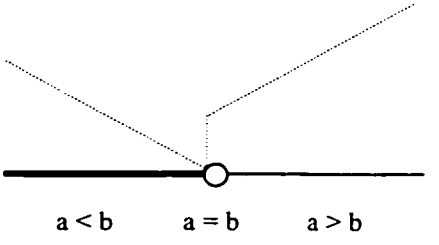
Case 2: .....



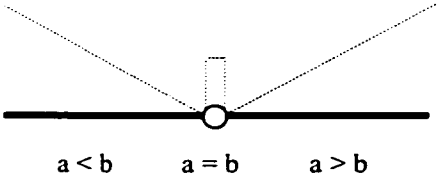
Case 3: .....



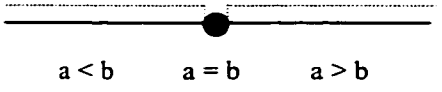
Case 4: .....



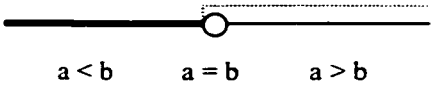
Case 5: .....



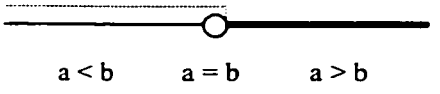
Case 6: .....



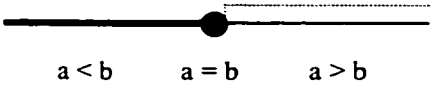
Case 7: .....



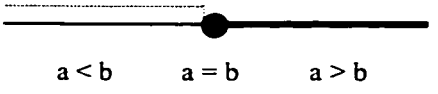
Case 8: .....



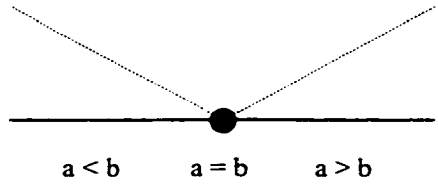
Case 9: .....



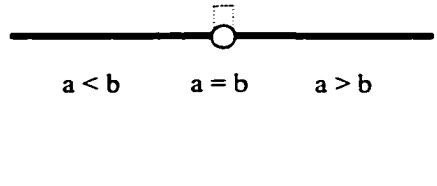
Case 10: .....



Case 11: .....



Case 12: .....



REFERENCES

[1] Balci, O., "Verification, Validation and Testing," *The Handbook of Simulation*, J. Banks, editor, John Wiley & Sons, New York, 1998.

[2] Frinke, D., Wolber, D., Fisher, G., Cohen, G., "Requirements Specification Language (RSL) and Supporting Tools," NASA Contractor Report 189700, July 1992.

[3] Levene, A., Mullery, G., "An Investigation of Requirement Specification Languages: Theory and Practice," *IEEE Computer* 15, pp. 50-59, May 1982.

[4] Davis, A., "The Design of a Family of Application-Oriented Requirements Languages," *IEEE Computer* 15, pp. 21-28, May 1982.

[5] Demetrovics, J., Knuth, E., Rado, P., "Specification Meta Systems," *IEEE Computer* 15, pp. 29-35, May 1982.

[6] Heitmeyer, C., Kirby, J., Labaw, B., "The SCR Method for Formally Specifying, Verifying and Validating Requirements: Tool Support," *Proceedings of the 1997 International Conference on Software Engineering*, pp. 610-611, Boston, May 1997.

[7] Awad, M., Kuusela, J., Ziegler, J., "Object-Oriented Techniques for Real-Time Systems: A Practical Approach Using OMT and Fusion," Prentice Hall, 1996.

[8] Potts, C., Takahashi, K., Anton, A., "Inquiry-Based Requirements Analysis," *IEEE Software*, 11, pp. 21-32, March 1994.

[9] Hsia, P., Samual, J., Gao, J., Kun, D., Toyoshima, Y., Chen, C., "Formal Approach to Scenario Analysis," *IEEE Software*, 11, pp. 33-41, March 1994.

[10] Hsia, P., Gao, J., Samuel, J., Kung, D., Toyoshima, Y., Chen, C., "Beharior-Based Acceptence Testing of Software Systems: A Formal Scenario Approach," *Proceedings of the Eighteenth Annual International Compeuter Software and Applications Conference (COMPSAC 94)*, pp. 293-298, Los Alamitos, CA, USA, 1994.

[11] Miller, T., Taylor, B., "A Requirements Methodology for Complex Real-Time Systems," *Proceedings of the International Symposium on Current Issues of Requirements Engineering Environments*, pp. 133-141, Kyoto, Japan, September 20-21, 1982.

[12] Lovengreen, H., Ravn, A., Rischel, H., "Design of Embedded Real-Time Ssytems: Developing a Method for Practical Software Engineering," *Proceedings of the IEEE International Conference on Computer Systems and Software Engineering*, pp. 385-390, 1990.

[13] White, S., "Comparative Analysis of Embedded Computer System Requirements Methods," *Proceedings of the First International Conference on Requirements Engineering*, pp. 126-134, 1994.

[14] Hsia, P., Davis, A., Kung, D., "Status Report: Requirements Engineering," *IEEE Software*, 10, pp. 75-79, November 1993.

[15] Davis, A., Hsia, P., "Giving Voice to Requirements Engineering," *IEEE Software*, 11, pp. 12-16, March 1994.

[16] Harel, D. "STATEMATE: A Working Environment for the Development of Complex Reactive Systems." *IEEE Transactions on Software Engineering*, 16(4), pp. 403-414, 1990.

[17] Heitmeyer C., Kirby, J., Labaw B., "Tools for Formal Specification, Verification, and Validation of Requirements," *Proceedings of the 12th Annual Conference on Computer Assurance (COMPASS'97)*, pp. 35-47, Gaithersburg, MD, June 1997.

[18] Hsia, P., Kung, D., Sell, C., "Software Requirements and Acceptance Testing," *Annals of Software Engineering*, vol. 3, pp. 291-317, 1997.

[19] Cunning, S.J., Ewing, T.C., Olson, J.T., Rozenblit, J.W., Schulz, S., "Towards an Integrated, Model-Based Codesign Environment," *Proceedings of the 1999 IEEE Conference and Workshop on Engineering of Computer Based Systems (ECBS'99)*, pp. 136-143, Nashville, TN, March 1999.

[20] Schulz, S., Rozenblit, J.W., Mrva, M. and Buchenrieder, K., "Model-Based Codesign," *IEEE Computer*, 31(8), 60-67, August 1998.

[21] Wolf, W., *Hardware-Software Co-Synthesis of Distributed Embedded Systems*, Boston, MA: Kluwer Academic Publishers, 1996.

[22] Ganssle, J., *The Art of Designing Embedded Systems*, Boston; Oxford: Newnes, 2000.

[23] Douglass, B., *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*, Reading, MA: Addison-Wesley, 1999.

[24] Chiodo, M., Giusto, P., Jurecska, A., Hsieh, H.C., Sangiovanni-Vincentelli, A., and Lavagno, L., "Hardware-Software Codesign of Embedded Systems," *IEEE Micro*, 14(4), pp. 26-36, 1994.

[25] Gajski, D., Narayan, S., Vahid, F., and Gong, J., *Specification and Design of Embedded Systems,* Englewood Cliffs, NJ: Prentice-Hall, 1994.

[26] Kumar, S., *A Unified Representation for Hardware/Software Codesign,* Ph.D. Dissertation, University of Virginia, 1995.

[27] De Micheli, G. and Gupta, R.K., "Hardware/Software Co-Design," *Proceedings of the IEEE*, 85(3), pp. 349-365, 1997.

[28] Rozenblit, J.W. and Buchenrieder, K. (Eds.), *Codesign: Computer-Aided Software/Hardware Engineering*, IEEE Press, 1994.

[29] Rozenblit, J.W., "Experimental Frame Specification Methodology for Hierarchical Simulation Modeling," *International Journal of General Systems*, 19(3), pp. 317-336, 1991.

[30] Zeigler, B.P. "Multifaceted Modeling and Discrete Event Simulation." Academic Press, London; Orlando, 1984.

[31] Zeigler, B.P., Praehofer, H., Kim, T.G., "Theory of Modeling and Simulation," 2nd Edition, Academic Press, 2000.

[32] Schulz, S. "A Model-Based Codesign Application: The Design of an Autonomous Intelligent Cruise Controller." Masters Thesis for the Department of Electrical & Computer Engineering, University of Arizona, Spring 1997.

[33] Schulz, S., Rozenblit, J.W., Buchenrieder, K., Mrva, M., "A Prototyping Environment for Model-Based Codesign," *Proceedings of the 1998 IEEE Conference and Workshop on Engineering of Computer Based Systems (ECBS'98)*, pp. 145 – 149, Maale Hachamisha, Isreal, March 30 – April 3, 1998.

[34] Goldsack, S., Finkelstein, A., "Requirements Engineering for Real-Time Systems," *software Engineering Journal*, 6(3), pp. 101-115, May 1991.

[35] Madnrioli, D., Morasca, S., Morzenti, A., "Generating Test Cases for Real-Time Systems from Logic Specifications," *ACM Transactions on Computer Systems*, 13(4), pp. 365-398, 1995.

[36] Morasca, S., Morzenti, A., SanPeitro, P., "Generating Functional Test Cases in-the-large for Time Critical Systems from Logic Specifications," *Proceedings of the 1996 International Symposium on Software Testing and Analysis*, pp. 39-52, Mission Beach, California, January 8-10, 1996.

[37] Ostroff, J., *Temporal Logic for Real-Time Systems*, New York, Wiley, 1989.

[38] Weber, R., Thelen, K., Srivastava, A., Krueger, J., "Automated Validation Test Generation," *Thirteenth AIAA/IEEE Digital Avionics Systems Conference (DASC94)*, pp. 99-104, November, 1994.

[39] Ho, I., Lin, J., "A Method of Test Cases Generation for Real-Time Systems," *Proceedings of the first International Symposium on Object-Oriented Real-Time Distributed Computing*, pp. 249-253, April 20-22, 1998.

[40] Chandrasekharan, M., Dasarathy, B., Kishimoto, Z., "Requirements-Based Testing of Real-Time Systems: Modeling for Testability," *IEEE Computer* 18, pp. 71-80, May 1985.

[41] Bauer, J., Finger, A., "Test Plan Generation Using Formal Grammars," *Proceedings of the Fourth International conference on Software Engineering*, pp. 425-432, Los Alamitos, California, September, 1979.

[42] Savor, T., Seviora, R., "Toward Automatic Detection of Software Failures," *IEEE Computer* vol. 31(8), pp. 68-74, August 1998.

[43] Chow, T., "Testing Software Design Modeled by Finite-State Machines," *IEEE Transactions on Software Engineering*, SE-4(3), pp. 178-187, 1978.

[44] Luo, G., Petrenko, A., Bochmann, G., "Selecting Test Sequences for Partially-Specified Nondeterministic Finite State Machines," *IFIP 7th International Workshop on Protocol test systems*, pp. 95-110, November, 1994.

[45] Gargantini, A., Heitmeyer, C., "Using Model Checking to Generate Tests from Requirements Specifications," *Proceedings of the Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT on Foundations of Software Engineering (ESEC/FSE99)*, pp. 146-162, Toulouse, France, September 6-10, 1999.

[46] Dalal, S., Jain, A., Patton, G., Rathi, M., Seymour, P., "AETG^{SM} Web: A Web Based Service for Automatic Efficient Test Generation from Functional Requirements," *Proceedings of the 2nd IEEE workshop on Industrial Strength Formal Specification Techniques*, pp. 84-85, October, 1998.

[47] Cunning, S.J., Rozenblit, J.W., "Test Scenario Generation from a Structured Requirements Specification," *Proceedings of the 1999 IEEE Conference and Workshop on Engineering of Computer Based Systems (ECBS'99)*, pp. 166-172, Nashville, TN, March 1999.

[48] Heitmeyer, C.L., Jeffords, R.D., Labaw, B.G., "Automated Consistency Checking of Requirements Specifications," *ACM Transactions on Engineering and Methodology*, vol. 5(3), pp. 231-261, July 1996.

[49] Heitmeyer, C., "Requirements Specifications for Hybrid Systems," Proceedings, *Hybrid Systems Workshop III*, Lecture Notes in Computer Science, Springer-Verlag, edited by R. Alur, T. Henzinger, and E. Sontag, 1996.

[50] Bharadwaj, R., Heitmeyer, C., "Verifying SCR Requirements Specifications Using State Exploration," *Proceedings of the First ACM SIGPLAN Workshop on Automatic Analysis of Software*, January 1997.

[51] Bharadwaj, R., Heitmeyer, C., "Hardware/Software Co-Design and Co-Validation Using the SCR Method," *IEEE International High Level Design Validation and Test Workshop (HLDVT'99)*, pp. 81-86, San Diego, November, 1999.

[52] Corman, T.H., Leiserson, C.E., Rivest, R.L., *Introduction to Algorithms*, The MIT Press, Cambridge Massachusetts, 1997.

[53] Cunning, S.J., Rozenblit, J.W., "Automatic Test Case Generation from Requirements Specifications for Real-time Embedded Systems," *Proceedings of the 1999 IEEE International Conference on Systems, Man, and Cybernetics (SMC'99)*, Vol. V, pp. 784-789, Tokyo, Japan, October 12-15, 1999.

[54] Courtios, P.J., Parnas, D.L., "Documentation for Safety Critical Software," *Proceedings of the 15th International Conference on Software Engineering (ICSE'93)*, pp. 315-323, Baltimore, MD, 1993.

[55] Gill, A., *Introduction to the Theory of Finite-State Machines*. McGraw-Hill, New York, 1962.

[56] Kohavi, Z., *Switching and Finite Automata Theory*, McGraw-Hill, New York, 1978.

[57] Deshmukh, R., Hawat, G., "An Algorithm to Determine Shortest Length Distinguishing, Homing, and Synchronizing Sequences for Sequential Machines," *Conference Record of Southcon '94*, pp. 496-501, March 1994.

[58] Zeigler, B., *Theory of Modeling and Simulation*. John Wiley & Sons, New York, 1976.

[59] Antsaklis, P., Michel, A., *Linear Systems*, McGraw-Hill, New York, 1997.

[60] Fujiwara, H., *Logic Testing and Design for Testability*, The MIT Press, Cambridge, Massachusetts, 1985.

[61] Liang, H., Chung, L., "Invalid State Identification for Sequential Circuit Test Generation," *Proceedings of the fifth Asian Test Symposium*, pp. 10-15, November, 1996.

[62] Tan, Q., Petrenko, A., Bochmann, G., "A Test Generation Tool for Specification in the Form of State Machines," *conference record of IEEE International Conference on Communications (ICC'96)*, vol. 1, pp. 225-229, June, 1996.

[63] *Standard Test Language for All Systems – Common/Abbreviated Test Language for All Systems*, IEEE Press, IEEE Std 716-1995.

[64] *IEEE Guide to the Use of the ATLAS Specification*, IEEE Press, IEEE Std 771-1998.

[65] Dasarathy, B., "Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them," *IEEE Transactions on Software Engineering*, SE-11(1), pp. 80-86, January 1985.

[66] Oram, A., Talbott, S., *Managing Projects with make*. O'Reilly & Associates, Inc., Cambridge MA, USA, 1993.

[67] Levine, J., Mason, T., Brown, D., *lex and yacc*, . O'Reilly & Associates, Inc., Cambridge MA, USA, 1995.

[68] Kernighan, B., Ritchie, D., *The C Programming Language*, Prentice Hall, Englewood Cliffs NJ, 1988.

[69] Chow, A., "Parallel DEVS: A Parallel, Hierarchical, Modular Modeling Formalism and its Distributed Simulator." *SCS Transactions on Simulation*, 13(2): pp. 55-102, 1996.

[70] Jaroch Gonzalez, Y. "Automated Validation of System Requirements for Embedded Systems Design" Masters Thesis for the Department of Electrical & Computer Engineering, University of Arizona, Spring 1999.

[71] Jaroch, Y., Cunning, S., Rozenblit, J.W., "Automated Validation of System Requirements for Embedded Systems Design," *Proceedings of the SCS Conference on AI, Simulation and Planning In High Autonomy Systems*, pp. 243-249, Tucson Arizona, March 6-8, 2000.

[72] Fröhlich, M., Werner, M., "Demonstration of the interactive Graph Visualization System daVinci," *Proceedings of DIMACS Workshop on Graph Drawing '94*, Princeton (USA) 1994, LNCS No. 894, Springer Verlag, 1995.

[73] Werner, M., "daVinci V2.1.x Online Documentation", Universität Bremen, http://www.tzi.de/~davinci/doc_V2.1/, June 1998.

[74] *daVinci Graph Visualization Tool Home Page*, Universität Bremen, http://www.tzi.de/davinci/

[75] Weinberg, G. M., "Rethinking Systems Analysis and Design," Little, Brown and Company, 1982.

[76] Winokur, M., Lavi, J., Lavi, I., Oz,. R., "Requirements Analysis and Specification of Embedded Systems Using ESCAM - a Case Study," *Proceedings of the 1990 International Conference on Computer Systems and Software Engineering*, pp. 80-89, 1990.

[77] Awad, M., Kuusela, J., Ziegler, J., "Object-Oriented Techniques for Real- Time Systems: A Practical Approach Using OMT and Fusion," Prentice Hall, 1996.

[78] Lovengreen, H., Ravn, A., Rischel, H., "Design of Embedded Real-Time Ssytems: Developing a Method for Practical Software Engineering," *Proceedings of the IEEE International Conference on Computer Systems and Software Engineering*, pp. 385-390, 1990.

[79] White, S., Lavi, J., "Embedded Computer System Requirements Workshop," *IEEE Computer* 18, pp. 67-70, April 1985.

[80] Siddiqi, J., "Challenging Universal Truths of Requirements Engineering," *IEEE Software*, 11, pp. 18-19, March 1994.

[81] Davis, A.M., *Software Requirements: Object, Functions, and States*, Prentice-Hall, New Jersey, 1993.

[82] Potts, C., Takahashi, K., Anton, A., "Inquiry-Based Requirements Analysis," *IEEE Software*, 11, pp. 21-32, March 1994.

[83] Hooper, J., Hsia, P., "Scenario-Based Prototyping for Requirements Identification," *Software Engineering Notes*, 7(5), pp. 88-93, December 1982.

[84] Ludewig, J., Boveri, B., "Computer-Aided Specification of Process Control Systems," *IEEE Computer* 15, pp. 12-20, May 1982.

[85] Miller, T., Taylor, B., "A Requirements Methodology for Complex Real- Time Systems," *Proceedings of the International Symposium on Current Issues of Requirements Engineering Environments*, pp. 133-141, Kyoto, Japan, September 20-21, 1982.

[86] Matsumoto, Y., Tanaka, T., Kawakita, S., "Specification Transformations and a Requirements Specification of Real-Time Control," *Proceedings of the International Symposium on Current Issues of Requirements Engineering Environments*, pp. 143-149, Kyoto, Japan, September 20-21, 1982.

[87] Buescher, T. W., Wilkinson, R. T., "Requirements modeling for real-time software development," *proceedings of the IEEE 1990 National Aerospace and Electronics Conference*, pp. 613-617, 1990.

[88] Jaffe, M., Leveson, N., Heimdahl, M., Melhart, B., "Software Requirements Analysis for Real-Time Process-Control Systems," *IEEE Transactions on Software Engineering*, 17(3), pp. 241-258, March 1991.

[89] Levene, A., Mullery, G., "An Investigation of Requirement Specification Languages: Theory and Practice," *IEEE Computer* 15, pp. 50-59, May 1982.

[90] Demetrovics, J., Knuth, E., Rado, P., "Specification Meta Systems," *IEEE Computer* 15, pp. 29-35, May 1982.

[91] Frinke, D., Wolber, D., Fisher, G., Cohen, G., "Requirements Specification Language (RSL) and Supporting Tools," NASA Contractor Report 189700, July 1992.

[92] Cohen, D., Feather, M. S., Narayanaswamy, K., Fickas, S. S., "Automatic Monitoring of Software Requirements," *Proceedings of the 1997 IEEE 19th International Conference on Software Engineering*, pp. 602-603, Boston, MA, May 1997.

[93] Frezza, S., "Automating Requirements-Based Testing for Hardware Design," RE'95 Doctoral Consortium, 1995.

[94] Clarke, D., Lee, I., "Automatic Generation of Tests for Timing Constraints from Requirements," *Proceedings of the 3rd International Workshop on Object-Oriented Real-Time Dependable Systems*, pp. 199-206, February, 1997.

[95] Clarke, D., Lee, I., "Automatic Test Generation for the Analysis of a Real-Time System: Case Study," *Proceedings of the third IEEE Real-Time Technology and Applications Symposium*, pp. 112-124, 1997.

[96] Ho, I., Lin, J., "A Method of Test Cases Generation for Real- Time Systems," *Proceedings of the first International Symposium on Object-Oriented Real-Time Distributed Computing*, pp. 249-253, April 20-22, 1998.

[97] En-Nouaary, A., Dssouli, R., Khendek, F., Elqortobi, A., "Timed Test Cases Generation Based on State Characterization Technique," *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS98)*, pp. 220-229, Madrid, Spain, December, 1998.

[98] En-Nouaary, A., Khendek, F., Dssouli, R. "Fault Coverage in Testing Real- Time Systems," *Procedings of the sixth IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA99)*, pp. 150-157, 1999.

[99] Ammann, P., Black, P., Majurski, W. "Using Model Checking to Generate Tests from Specifications," *Proceedings of the 2nd IEEE International Conference on Formal Engineering Methods (ICREM'98)*, pp. 46-54, Brisbane, Australia, December, 1998.

[100] Glover, T., Cardell-Oliver, R., "A Modular Tool for Test Generation for Real-Time Systems," *IEE Colloquium. Applicable Modelling, Verification and Analysis Techniques for Real-Time Systems*, pp. 3/1-4, London, UK, 1999.