
*Journal of Intelligent and Robotic Systems* **41:** 87–112, 2004.



# Automating Test Generation for Discrete Event Oriented Embedded Systems

STEVEN J. CUNNING and JERZY W. ROZENBLIT
*University of Arizona, Electrical & Comp. Eng. Department, P.O. Box 210104, 1230 E. Speedway Boulevard, Tuscon, AZ 85721-0104, USA; e-mail: head@ece.arizona.edu*

**Abstract.** A method for the automatic generation of test scenarios from the behavioral requirements of a system is presented in this paper. The generated suite of test scenarios validates the system design or implementation against the requirements. The approach proposed here uses 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. Measurements of several metrics on the scenario generation algorithms have been collected using prototype tools.

**Key words:** test pattern generation, requirements modeling, embedded systems.

## 1. Introduction

### 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, 16, 17, 22, 30, 33, 35, 37–40]. White [45] 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.

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 [1, 3, 6, 7, 24, 26, 33–35], 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).

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 [34]. 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 [12, 43]. 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 they 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. RELATED WORK

In [11], we provide an extensive survey of publications related to our work. Here we present a brief synopsis of previous work.

Heitmeyer et al. describe the SCR formalism in [29]. 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. In [30, 31] they describe tool support for the SCR language. A set of tools is available to support the development, automatic consistency checking, symbolic simulation, and dependency graph browsing, of SCR models.

In the area of test generation from state-based specifications, arguably the defining work is by Chow [7]. Chow describes a method to test the control structure of software that can be modeled by finite state machines. 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).

Hsia et al. provides a good treatment of scenario analysis in [33–35]. 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.

Clarke and Lee [8, 9] 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).

Ho and Lin [32] 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. Glover and Cardell-Oliver [26] 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.

In [5] a method for generating conformance tests for real-time systems from timed automata specifications is presented. The goal of the generated test suite is to show trace equivalence between the system specification and the implementation. Views are defined in an attempt to limit size complexity. Views define the aspects of the system that are important to the particular test objective. Test cases are derived from traces of the specification after expanding all reachable states. The test suite is comprised of tests for every transition in the specification. Distinguishing sequences are appended as needed to distinguish final states that are within the same equivalence class due to the selected view. Time is handled by quantizing system clocks and including the clocks as part of the system state. For behavior that is acceptable within a temporal range, timed transitions for all quantized time values within the range are included in the specification. This leads to observable nondeterminism requiring the generation of all possible traces to support comparison of test results in some cases.

While many of the previous references are closely related to the approach presented in our 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) [1, 32], finite state machine descriptions (often augmented) [3, 6, 7, 20, 35], timed Petri nets [32], rule-based models [44], input/output specifications [15], the algebra of communicating resources [8], state-based models [1, 5, 24, 26], and more general languages such as the Requirements Specification Language (RSL) [21].

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 [7], En-Nouaary et al. [20], and

Cardell-Oliver [5] 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 [21] 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 [1, 3, 7, 15, 21, 24, 35, 44]. For real-time systems, temporal correctness is equally important. Temporal correctness is addressed by the tests generated in [5, 6, 8, 20, 26, 32]. Our treatment of temporal requirements is described in [11].

### 1.3. 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. More specifically, this work focuses on embedded computer-based systems with hard real-time deadlines [19, 23, 46]. Their development is often supported by codesign techniques [42].

The variant of codesign under development at the University of Arizona has been termed *model-based codesign* [12, 43]. In model-based codesign, we verify correctness of models through computer simulation. A simulation test setup is called an *experimental frame* [41, 47, 49] and 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. The test scenarios generated by the algorithms described in this paper are intended to support the testing needs of all levels of the model-based codesign process.

## 2. Test Generation Process

The top level 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 goal is to automate the defined process to the greatest extent possible. In order to facilitate automation, a set of algorithms has been developed for certain steps in the test generation process.

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.

## 2.1. TEST GENERATION OVERVIEW

The proposed test generation process proposed to support model-based codesign is illustrated in Figure 1. 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 specified in a formal modeling language. 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

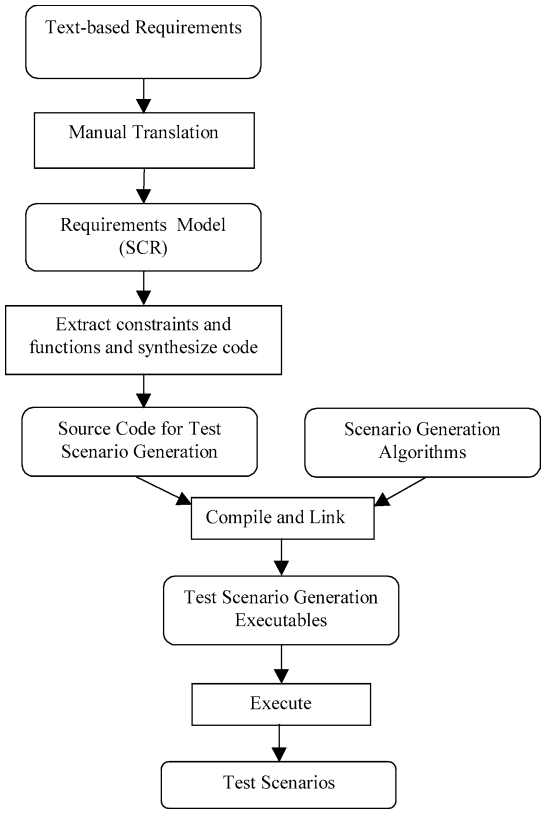


*Figure 1.* Scenario generation and use.

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 as described in [27]. This code is compiled with the test scenario generation algorithms 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.

## 3. Example System

The example system is a Safety Injection System for a nuclear reactor adapted from [10]. A block diagram of the system is shown in Figure 2. 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*.

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).

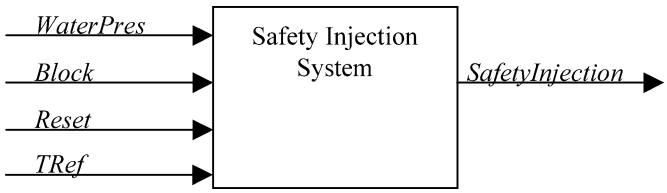


*Figure 2.* Safety Injection System block diagram.

*Table I.* 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* |

*Table II.* Mode transition table for pressure

| Old mode | Event | New mode | |
|---|---|---|---|
| TooLow | @T(*WaterPres* $\geqslant$ LOW) | Permitted | [P1] |
| Permitted | @T(*WaterPres* $<$ LOW) | TooLow | [P2] |

*Table III.* 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 | |

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

The requirements model will be illustrated in the Software Cost Reduction (SCR) formalism [29]. 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, @C($A$) is the event that A has changed, and a variable in *primed* notation (e.g., $A'$) indicates the value that $A$ will assume in the new state. The four tables (Tables II–V) define the functions that control the values of *Pressure*, *Overridden*, *TRefCnt*, and *SafetyInjection*, respectively.

*Table IV.* Event table for *TRefCnt*

| Mode | Events | | | |
|---|---|---|---|---|
| TooLow | @C(*TRef*) when (*Overridden*) | [R4a] | @C(*Block*) when (*Reset* = Off) | [R4c] |
| *TRefCnt*′ | *TRefCnt* + 1 | | 0 | |

*Table V.* 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.

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. Part 1: Base Test Scenario Generation

We have defined the test generation problem in terms of a covering of the requirements. 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 exercises all state transitions associated with all uniquely identified system requirements. 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.

It must be noted that there is not a one-to-one mapping of textual requirements to actions in the formal requirements model. The process of developing the re-

quirements model is the conversion of the textual requirements into the formal SCR 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 or multiple transitions. 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. A unique requirement identifier will be associated with every action in the requirements model.

### 4.1. 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). The cost is the increased complexity of solving the problem. It has been shown in [11] that the requirements covering problem is NP-complete.

### 4.2. HEURISTIC APPROACH TO SOLVING THE REQUIREMENTS COVERING PROBLEM

Given the hardness of the requirements covering problem, it is appropriate to seek an approximation or heuristic approach that will provided an adequate solution in a reasonable amount of time.

Initially, no requirements will have been covered, thus any available stimulus that causes a state change in the model will help reach the covering 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 a scenario search tree, then add the state connected by the edge covering the most requirements not yet covered. The number of non-covered 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.

The greedy approach is not a complete solution because what could eventually happen, and is 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 yet 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 state

space of the requirements model. 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 exercised through a long series of inputs in order to get the system into a state that allows the needed transition to be taken. The remaining problem is how to guide the expansion process so that it is likely to follow a path leading to a needed transition.

If the complete scenario search tree already existed, the states connected to a needed 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 derived (most likely with *don't care* values) that will allow the desired transition to be taken. The selection of the next state to expand, 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 the *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 [11, 13]. 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*.

Base scenarios are limited in that they are not suitable for *black box* testing. Each test scenario generated forces the system to traverse a set of state transitions. Unfortunately the base 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. Based on these objectives, the base scenarios are incomplete.

PART 1 ALGORITHM.

1. Starting from the specified initialization state, perform a greedy state search of the requirements model by applying all available inputs to the currently selected state. Any transitions covering requirement IDs not previously covered are added to the scenario tree. The next state to expand is the state reached through the transition covering the most requirement IDs not previously covered.
2. If all requirement IDs are not covered, repeatedly select one of the remaining requirement IDs as a target until all remaining requirement IDs have been selected, and then perform a distance based search using the leaf states expanded by step 1 as the starting pool of states available to expand. States are expanded as before, by applying all available inputs to the currently selected state. Any transitions covering a requirement ID that has not previously been covered, along with the sequence of transitions that connect this transition to the scenario tree, are added to the scenario tree. As new states are expanded, they are added to the pool of available states. The next state to expand is the available state with the smallest distance value.
3. Output, as an intermediate result, all paths from the initialization state (i.e. root node) to a leaf node in the scenario tree. These sequences of stimulus, response, and state data define the base scenarios.

### 4.3. PART 1 ALGORITHM EXAMPLE

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 3. 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 VI. 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.

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 3. 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.

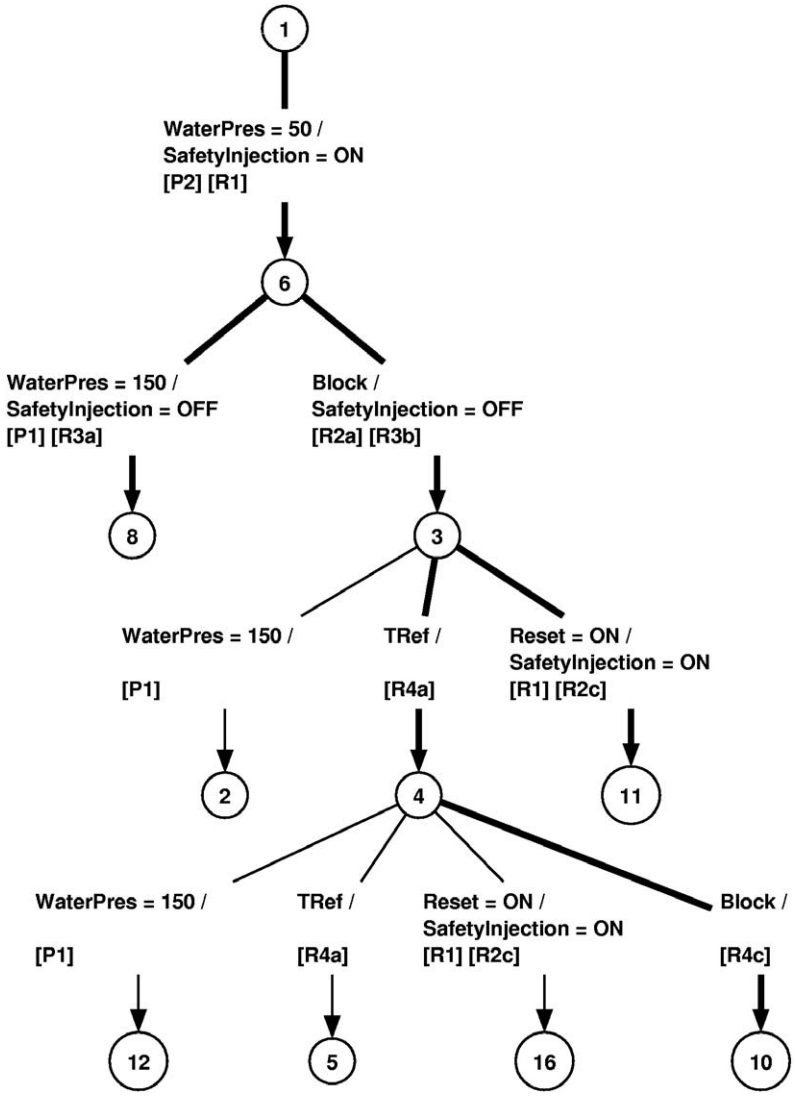


*Figure 3.* Greedy scenario search tree for Safety Injection System.

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 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

*Table VI.* 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 |

*Table VII.* 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 |

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 VII.

The base scenario search tree at the completion of the Part 1 Algorithm is given in Figure 4. Again, the scenario tree *ST* is represented by the bold edges and defines the base scenarios.

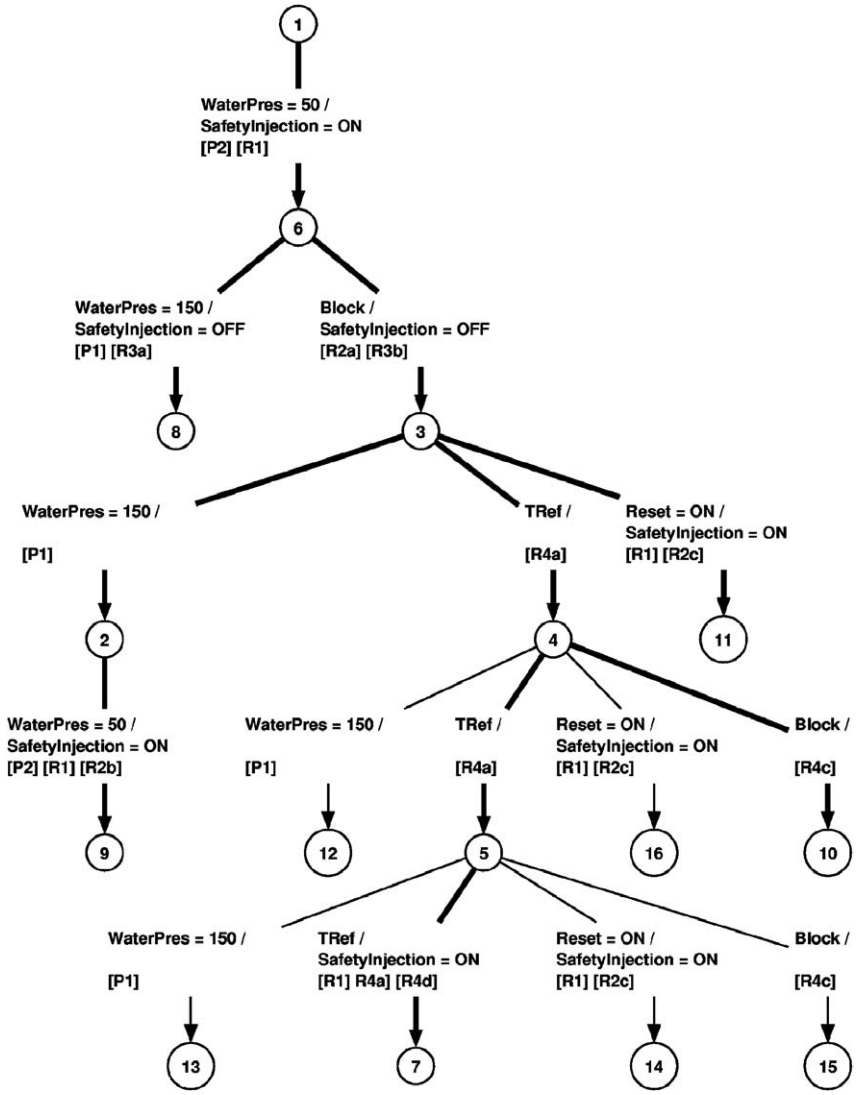


*Figure 4.* Final base scenario search tree for Safety Injection System.

## 5. Part 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 action associated with the target requirement identifier. 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.

PART 2 ALGORITHM.

1. For each step of each base scenario and each requirement ID exercised by that step that has not been identified as verifiable, simulate the requirement model with and without the transition associated with the current requirement ID to the end of the current scenario or until a difference is detected at a system output. If a difference was detected, mark the requirement ID as verifiable.
2. Output, as an intermediate result, the list of requirement IDs not marked as verifiable.

### 5.1. PART 2 ALGORITHM EXAMPLE

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, R4c, as not being verifiable for black box testing. Requirement R4c is exercised in the last transition of scenario 4, which is defined by the transitions from state 1 to state 10 in Figure 4. R4c resets the value of *TRefCnt* to zero when *Pressure* is TOOLOW and *Reset* is not On. The previous step in this scenario 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.

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 with 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.

## 6. Part 3: Enhancing the Base Scenarios

After the incomplete base test scenarios are identified, they must be enhanced to support 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 [18, 25, 36, 48]. In [11], we provide a detailed discussion on how various authors have addressed the distinguishability problem. Here, we introduce a heuristic called *difference-based search* that enables us to enhance the base scenarios.

Starting with the premise that all internal change (IC) transitions (state changes that are not immediately observable at a system output) 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 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 transition 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 the 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.

PART 3 ALGORITHM.

1. For each requirement ID identified by Part 2 as unverified, perform a difference based search starting with a single state pair. State pairs are generated by simulating the requirements model with and without the transition associated with the current requirement ID disabled. The initial state pair is defined by the states reached after applying the base scenario stimulus for exercising the current requirement ID. As states are expanded, new state pairs that contain a difference are added to the pool of available states. The search continues until a difference is detected at a system output or the pool of available states is empty.
2. If a difference was detected at a system output, as an intermediate result, output the successful path from the generated search tree (sequence of stimulus, response, and state data).

### 6.1. PART 3 ALGORITHM EXAMPLE

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 5 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 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 to On one *TRef* event too soon, which is indicated by the "Good" and "Bad" values listed above.

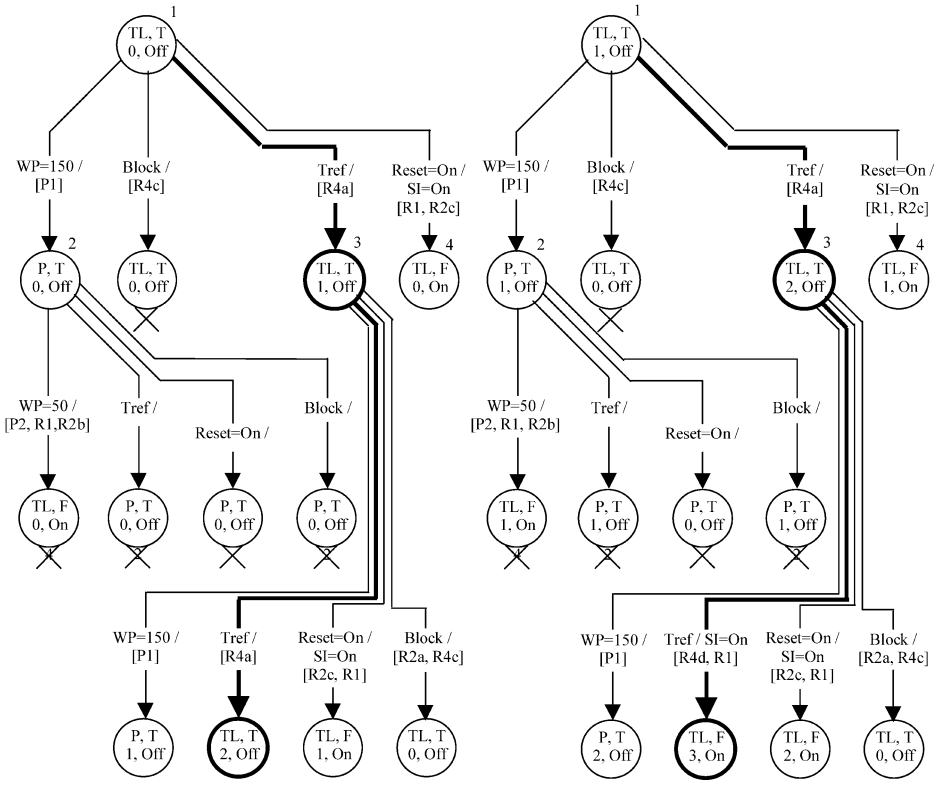


*Figure 5.* Application of difference-based search.

## 7. Part 4: Combining Base 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 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.

Additional details regarding the test generation algorithms are presented in [11].

PART 4 ALGORITHM.

1. Merge the base scenarios from part 1 with the scenario enhancements from Part 3 by appending the scenario enhancements to the base scenario tree. Before adding a state from an enhancement, check for equivalent states in the base scenario tree to detect overlapping paths and avoid redundancy.
2. Output all paths from the tree resulting from the previous step. These sequences of stimulus, response, state data represent the generated test suite.

### 7.1. PART 4 ALGORITHM EXAMPLE

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 above, 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 6. 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

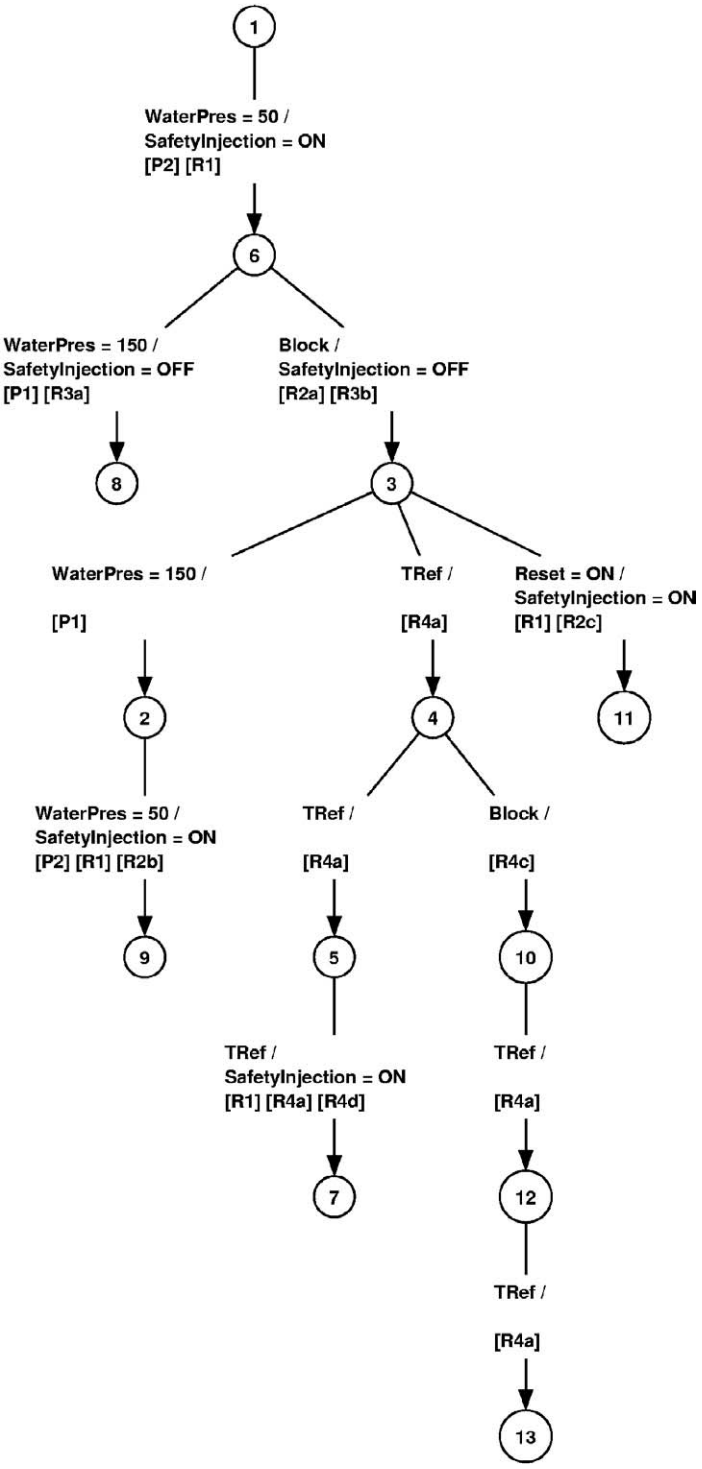


*Figure 6.* Final scenario tree for Safety Injection System.

states and edges define scenarios, rather than a scenario search tree which includes the results of unproductive search. This tree defines the final set of test scenarios for the Safety Injection System.

## 8. Experimental Results

This section reports on the results collected while experimenting with the implemented test generation algorithms. Several systems have been used to evaluate the test generation process. The Safety Injection System (SIS) that has been used throughout the preceding sections 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 [4, 28, 29] 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 [24]. 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 [24]. "Negative action" transitions do not change the state of the system (i.e., they are equivalent to self loops in a state transition diagrams) and correspond to requirements that state what the system must not do.

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 [14]. This model differs from the others in that it was not implemented in SCR followed by synthesizing code. It was developed directly in C from the requirements forms proposed in [14].

The final system modeled is that of an elevator controller. This model represents a controller for an elevator in a three story building.

Table VIII provides a list of the models used and a the number of requirement identifiers for each.

*Table VIII.* Summary of modeled systems

| Name | No. of requirement IDs |
|---|---|
| SIS | 11 |
| SISsm | 10 |
| SISlg | 10 |
| SISsmNA | 24 |
| SISlgNA | 24 |
| Temperature controller | 31 |
| Elevator controller | 63 |

### 8.1. TEST SCENARIO GENERATION RESULTS

The test generation process of Figure 1 has been applied to each of the models described in the preceding section. Except for the development of the requirements model from the textual requirements, this process has been fully automated by the prototype tools described in [11, 27].

In what follows, we focus on two groups of results: (a) verifiabilty of requirements at the black-box level and (b) test generation efficiency.

### 8.2. REQUIREMENTS VERIFIABLE AT THE BLACK BOX LEVEL

As illustrated in Figure 7, 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.

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 the 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.

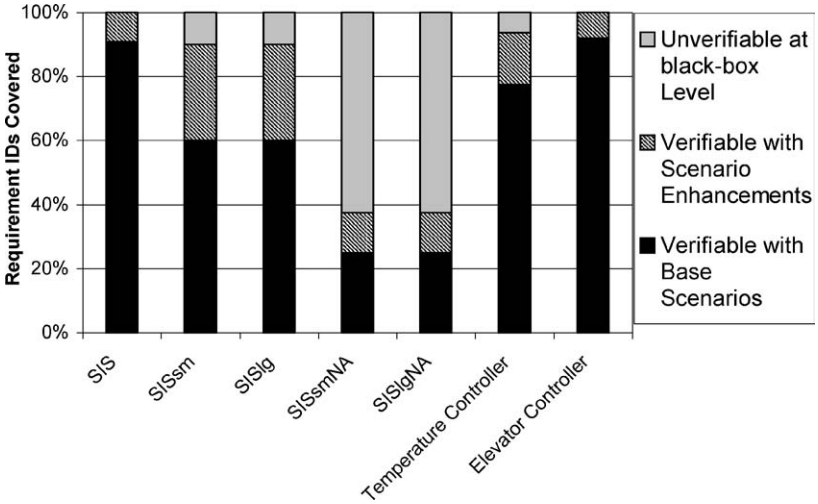


*Figure 7.* Black box verifiable requirements.

### 8.3. SCENARIO GENERATION EFFICIENCY

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.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.

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 8 charts the total execution times and a best fit curve. 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.

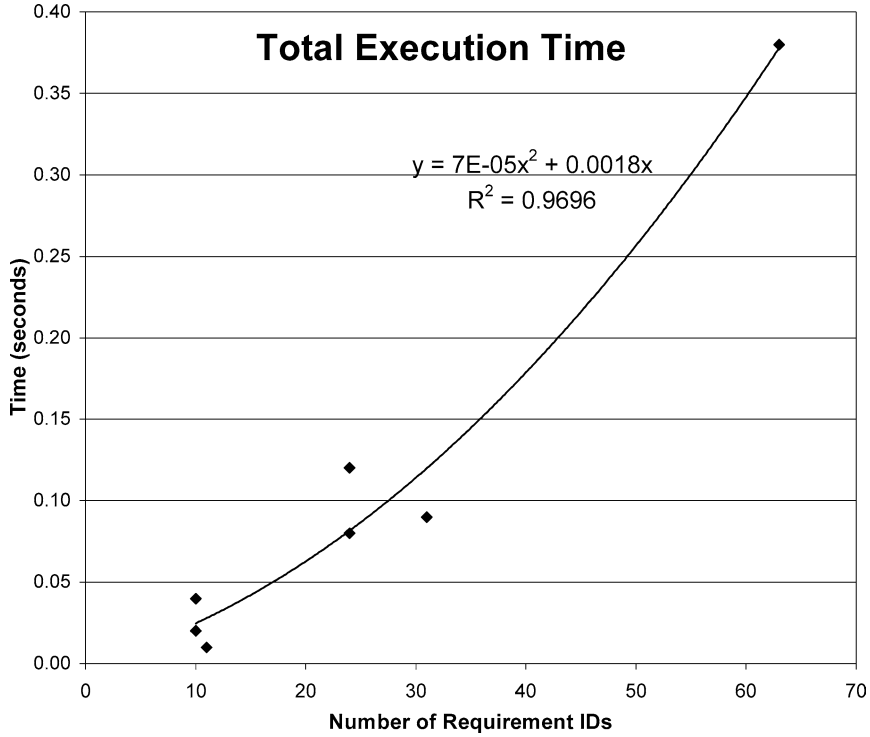


*Figure 8.* Total execution time.

## 9. Summary of Results

The primary research goal was defined to be the development of 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 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.

One of our primary goals was to avoid expanding the entire state space of the requirements model in order to define a practical test generation algorithm/method. Cardell-Oliver's method is likely to be computationally expensive in terms of both test generation time and test execution time (number of test cases).

We do not generate tests to demonstrate equivalence between the requirement model and units under test by testing every unique transition in the requirement model. Our goal is to exercise (cover) every state variable change defined (and identified by a unique requirement ID) in the requirements model. An implementation does not need to be trace equivalent to fulfill its requirements. What must be demonstrated is that the outward (black box) behavior of the implementation must fulfill the requirements. The test suite our method generates is sufficient for this.

Both methods, i.e., ours and that of Cardell-Oliver, generate test cases based on state space exploration of a requirements model/system specification and both append distinguishing sequences to allow conformance to be observed using a black box model of testing.

Both methods may not detect extra states in the system under test. We agree with the discussion and suggested approach in [5] for dealing with this issue. As presented, our algorithms do not address temporal aspects. The approach in [5], while technically sound, adds significant complexity to the specification and is likely to be difficult for practicing engineers. Our approach is to use an untimed requirements model, vary the temporal positioning of the environmental stimulus within the defined constraints, and to test the responses of the system under test for adherence to the system's temporal requirements.

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.

## Acknowledgements

This work was supported by grant number 9554561 from the National Science Foundation. Our thanks goes to the members of the Naval Research Labs, Constance Heitmeyer, Bruce Labaw, Jim Kirby, Ralph Jeffords, and Todd Grimm, for providing their SCR Toolset. We also thank the University of Bremen for the use of the graph visualization tool, DaVinci, which was used to create the graphs in figures 3, 4, and 6.

## References

1. Ammann, P., Black, P., and Majurski, W.: Using model checking to generate tests from specifications, in: *Proc. of the 2nd IEEE Internat. Conf. on Formal Engineering Methods (ICREM'98)*, Brisbane, Australia, December 1998, pp. 46–54.
2. Awad, M., Kuusela, J., and Ziegler, J.: *Object-Oriented Techniques for Real-Time Systems: A Practical Approach Using OMT and Fusion*, Prentice-Hall, Englewood Cliffs, NJ, 1996.
3. Bauer, J. and Finger, A.: Test plan generation using formal grammars, in: *Proc. of the Fourth Internat. Conf. on Software Engineering*, Los Alamitos, CA, September 1979, pp. 425–432.
4. Bharadwaj, R. and Heitmeyer, C.: Verifying SCR requirements specifications using state exploration, in: *Proc. of the First ACM SIGPLAN Workshop on Automatic Analysis of Software*, January 1997.
5. Cardell-Oliver, R.: Conformance tests for real-time systems with timed automata specifications, *Formal Aspects Comput.* (2000), 350–371.
6. Chandrasekharan, M., Dasarathy, B., and Kishimoto, Z.: Requirements-based testing of real-time systems: Modeling for testability, *IEEE Computer* **18** (May 1985), 71–80.
7. Chow, T.: Testing software design modeled by finite-state machines, *IEEE Trans. Software Engrg.* **4**(3) (1978), 178–187.
8. Clarke, D. and Lee, I.: Automatic generation of tests for timing constraints from requirements, in: *Proc. of the 3rd Internat. Workshop on Object-Oriented Real-Time Dependable Systems*, February 1997, pp. 199–206.
9. Clarke, D. and Lee, I.: Automatic test generation for the analysis of a real-time system: Case study, in: *Proc. of the Third IEEE Real-Time Technology and Applications Symposium*, 1997, pp. 112–124.
10. Courtios, P. J. and Parnas, D. L.: Documentation for safety critical software, in: *Proc. of the 15th Internat. Conf. on Software Engineering (ICSE'93)*, Baltimore, MD, 1993, pp. 315–323.
11. Cunning, S. J.: Automating test generation for discrete event oriented real-time embedded systems, PhD Dissertation, Department of Electrical & Computer Engineering, University of Arizona, 2000.
12. Cunning, S. J., Ewing, T. C., Olson, J. T., Rozenblit, J. W., and Schulz, S.: Towards an integrated, model-based codesign environment, in: *Proc. of the 1999 IEEE Conf. and Workshop on Engineering of Computer-Based Systems (ECBS'99)*, Nashville, TN, March 1999, pp. 136–143.
13. Cunning, S. J. and Rozenblit, J. W.: Automatic test case generation from requirements specifications for real-time embedded systems, in: *Proc. of the 1999 IEEE Internat. Conf. on Systems, Man, and Cybernetics (SMC'99)*, Vol. V, Tokyo, Japan, 12–15 October 1999, pp. 784–789.
14. Cunning, S. J. and Rozenblit, J. W.: Test scenario generation from a structured requirements specification, in: *Proc. of the 1999 IEEE Conf. and Workshop on Engineering of Computer-Based Systems (ECBS'99)*, Nashville, TN, March 1999, pp. 166–172.
15. Dalal, S., Jain, A., Patton, G., Rathi, M., and Seymour, P.: AETG$^{SM}$ Web: A Web-based service for automatic efficient test generation from functional requirements, in: *Proc. of the 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*, October 1998, pp. 84–85.

16. Davis, A.: The design of a family of application-oriented requirements languages, *IEEE Computer* **15** (May 1982), 21–28.
17. Demetrovics, J., Knuth, E., and Rado, P.: Specification meta systems, *IEEE Computer* **15** (May 1982), 29–35.
18. Deshmukh, R. and Hawat, G.: An algorithm to determine shortest length distinguishing, homing, and synchronizing sequences for sequential machines, in: *Conf. Record of Southcon'94*, March 1994, pp. 496–501.
19. Douglass, B.: *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*, Addison-Wesley, Reading, MA, 1999.
20. En-Nouaary, A., Dssouli, R., Khendek, F., and Elqortobi, A.: Timed test cases generation based on state characterization technique, in: *Proc. of the 19th IEEE Real-Time Systems Symposium (RTSS'98)*, Madrid, Spain, December 1998, pp. 220–229.
21. Frezza, S.: Automating requirements-based testing for hardware design, RE'95 Doctoral Consortium, 1995.
22. Frinke, D., Wolber, D., Fisher, G., and Cohen, G.: Requirements Specification Language (RSL) and supporting tools, NASA Contractor Report 189700, July 1992.
23. Ganssle, J.: *The Art of Designing Embedded Systems*, Boston, Oxford, Newnes, 2000.
24. Gargantini, A. and Heitmeyer, C.: Using model checking to generate tests from requirements specifications, in: *Proc. of the Joint 7th European Software Engineering Conf. and 7th ACM SIGSOFT on Foundations of Software Engineering (ESEC/FSE'99)*, Toulouse, France, 6–10 September 1999, pp. 146–162.
25. Gill, A.: *Introduction to the Theory of Finite-State Machines*, McGraw-Hill, New York, 1962.
26. Glover, T. and Cardell-Oliver, R.: A modular tool for test generation for real-time systems, in: *IEE Colloquium on Applicable Modelling, Verification and Analysis Techniques for Real-Time Systems*, London, UK, 1999, pp. 3/1–4.
27. Gupta, P., Cunning, S., and Rozenblit, J. W.: Synthesis of high-level requirements models for automatic test generation, in: *Proc. of the 2001 IEEE Conf. and Workshop on Engineering of Computer-Based Systems (ECBS'01)*, Washington, DC, April 2001, pp. 76–82.
28. Heitmeyer, C.: Requirements specifications for hybrid systems, in: R. Alur, T. Henzinger and E. Sontag (eds), *Proc. of Hybrid Systems Workshop III*, Lecture Notes in Computer Science, Springer, New York, 1996.
29. Heitmeyer, C. L., Jeffords, R. D., and Labaw, B. G.: Automated consistency checking of requirements specifications, *ACM Trans. Engrg. Methodology* **5**(3) (July 1996), 231–261.
30. Heitmeyer, C., Kirby, J., and Labaw, B., The SCR method for formally specifying, verifying and validating requirements: Tool support, in: *Proc. of the 1997 Internat. Conf. on Software Engineering*, Boston, May 1997, pp. 610–611.
31. Heitmeyer C., Kirby, J., and Labaw B.: Tools for formal specification, verification, and validation of requirements, in: *Proc. of the 12th Annual Conf. on Computer Assurance (COMPASS'97)*, Gaithersburg, MD, June 1997, pp. 35–47.
32. Ho, I. and Lin, J.: A method of test cases generation for real-time systems, in: *Proc. of the First Internat. Symposium on Object-Oriented Real-Time Distributed Computing*, 20–22 April 1998, pp. 249–253.
33. Hsia, P., Gao, J., Samuel, J., Kung, D., Toyoshima, Y., and Chen, C.: Behavior-based acceptence testing of software systems: A formal scenario approach, in: *Proc. of the Eighteenth Annual Internat. Computer Software and Applications Conf. (COMPSAC'94)*, Los Alamitos, CA, USA, 1994, pp. 293–298.
34. Hsia, P., Kung, D., and Sell, C.: Software requirements and acceptance testing, *Ann. Software Engrg.* **3** (1997), 291–317.
35. Hsia, P., Samual, J., Gao, J., Kun, D., Toyoshima, Y., and Chen, C.: Formal approach to scenario analysis, *IEEE Software* **11** (March 1994), 33–41.
36. Kohavi, Z.: *Switching and Finite Automata Theory*, McGraw-Hill, New York, 1978.

37. Levene, A. and Mullery, G.: An investigation of requirement specification languages: Theory and practice, *IEEE Computer* **15** (May 1982), 50–59.
38. Lovengreen, H., Ravn, A., and Rischel, H.: Design of embedded real-time systems: Developing a method for practical software engineering, in: *Proc. of the IEEE Internat. Conf. on Computer Systems and Software Engineering*, 1990, pp. 385–390.
39. Miller, T. and Taylor, B.: A requirements methodology for complex real-time systems, in: *Proc. of the Internat. Symposium on Current Issues of Requirements Engineering Environments*, Kyoto, Japan, 20–21 September 1982, pp. 133–141.
40. Potts, C., Takahashi, K., and Anton, A.: Inquiry-based requirements analysis, *IEEE Software* **11** (March 1994), 21–32.
41. Rozenblit, J. W.: Experimental frame specification methodology for hierarchical simulation modeling, *Internat. J. General Systems* **19**(3) (1991), 317–336.
42. Rozenblit, J. W. and Buchenrieder, K. (eds): *Codesign: Computer-Aided Software/Hardware Engineering*, IEEE Press, New York, 1994.
43. Schulz, S., Rozenblit, J. W., Mrva, M., and Buchenrieder, K.: Model-based codesign, *IEEE Computer* **31**(8) (1998), 60–67.
44. Weber, R., Thelen, K., Srivastava, A., and Krueger, J.: Automated validation test generation, in: *Thirteenth AIAA/IEEE Digital Avionics Systems Conf. (DASC'94)*, November 1994, pp. 99–104.
45. White, S.: Comparative analysis of embedded computer system requirements methods, in: *Proc. of the First Internat. Conf. on Requirements Engineering*, 1994, pp. 126–134.
46. Wolf, W.: *Hardware-Software Co-Synthesis of Distributed Embedded Systems*, Kluwer Academic Publishers, Boston, MA, 1996.
47. Zeigler, B. P.: *Multifaceted Modeling and Discrete Event Simulation*, Academic Press, London, 1984.
48. Zeigler, B.: *Theory of Modeling and Simulation*, Wiley, New York, 1976.
49. Zeigler, B. P., Praehofer, H., and Kim, T. G.: *Theory of Modeling and Simulation*, 2nd edn, Academic Press, New York, 2000.