Automatic Test Case Generation from Requirements Specifications for Real-time Embedded Systems

S.J. Cunning and J.W. Rozenblit
Department of Electrical and Computer Engineering,
The University of Arizona
Tucson, Arizona 85721-0104, U.S.A.
{scunning|jr}@ece.arizona.edu

ABSTRACT

This paper presents continuing research toward automatic generation of test cases from requirements specifications for event-oriented, real-time embedded systems. The requirements documentation and test case generation activities make up the initial steps in our method to realize model-based codesign [1]. In this codesign method, test cases are used to validate system models and prototypes against the requirements specification. This ensures coherence between the system models at various levels of detail, the system prototype, and the final system design. Automating the test case generation process provides a means to ensure that the test cases have been derived in a consistent and objective manner and that all system requirements have been covered. The formulation and difficulty of the test case generation problem are discussed and a heuristic algorithm to automatically generate test cases is presented. The inputs to the algorithm are extracted from the requirements specification. The algorithm is a two phase exploration of the system states defined in the requirements specification. The goal is to generate a suite of test cases that provide complete coverage of all documented system requirements. A design example is presented that is used to illustrate the generation of test-cases.

1. INTRODUCTION AND MOTIVATION

While the idea of automatic generation of test cases from requirements specifications is not new, acceptance of the proposed methods have not been embraced in industry. All such methods rely on the development of a finite state machine (FSM) based representation of the system requirements, 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 industry management must believe that the benefits of the requirements model outweighs 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 [2] and the SCR tool set [3].

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 tool support that is flexible and easily integrated with the requirements modeling tools is available. The automatic test generation must provide effective test cases 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 [4]. One of the benefits of automation is to provide consistent coverage of system requirements and to generate test cases 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. The time needed for test generation will be reduced which will allow an organization 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 [1,5]. 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.

In section 2 an overview of the test case generation process and how the generated tests are applied within Model-based Codesign are discussed. Section 3 describes the problem of covering all system requirements, determines why this problem is computationally hard, and proposes a heuristic approach to provide an adequate solution. Section 4 gives an example that is used to illustrate the generation of test cases using the heuristic approach. Section 5 concludes with a summary of related work and the future direction of our research.

2. TEST CASE GENERATION AND USE

Model-based codesign relies heavily on modeling and simulation to evaluate potential system designs. Efficient use of simulation requires prepared sets of system stimulus and a means for evaluating the system responses to them. Test cases must consist of a time ordered sequence of events representing both the stimulus to be applied to the system and the expected responses (also referred to as test scenario). The approach to test case generation that we propose 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, 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 prototype of the system that can be used to gain insight into the dynamic behavior of the system. This can be invaluable in correcting and clarifying the requirements.

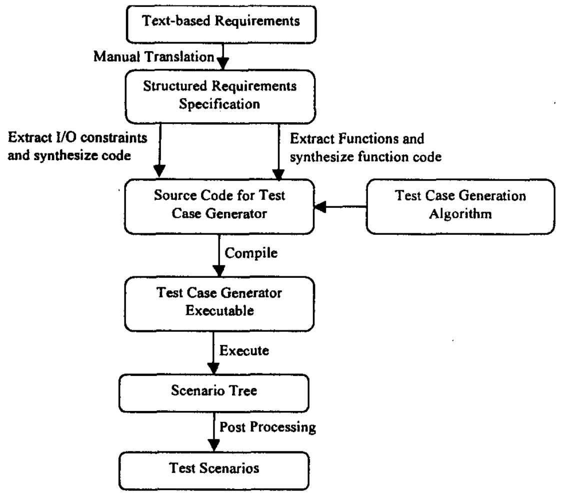
Text-based Requirements
Manual Translation
Structured Requirements Specification
Extract I/O constraints and synthesize code
Extract Functions and synthesize function code
Source Code for Test Case Generator
Test Case Generation Algorithm
Compile
Test Case Generator Executable
Execute
Scenario Tree
Post Processing
Test Scenarios

Figure 1. Test Case Generation Process

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 later used by the test case generator in order 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 code is compiled with the test case generation algorithm to produce the test case generator. The test case generator uses the system functions and interface constraints 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 stimulus in order to generate new system states. The control of the state exploration is described in section 3.

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

Application of test cases to design models is performed through the concept of an *experimental frame* [6]. The *experimental frame* is a formalism that ensures the clear separation between the model under test (MUT) and the environment. The *experimental frame* 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.

When the design proceeds to the prototype level, the same set of test cases will be used and interpreted by a real-time test environment.

3. PROBLEM FORMULATION

In the approach to test case generation presented in [7], the complete scenario tree is generated. This complete tree represents all possible system state transitions under the limitations of environmental constraints regarding the applications of stimulus and defined test values for stimulus containing data values. While this approach is practical for smaller systems, the tree may quickly grow beyond a manageable size for more complex systems.

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 this goal and the state explosion problem, it is appropriate to explore the possibility of applying a more 'intelligent' tree generation approach. Only a sufficient subset of the scenario tree should be generated such that when all scenarios from this subset are applied to the system, all system requirements are covered. In order to accomplish this, the test case generation algorithm will have to be modified from that described in [7] 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 define the problem at hand, 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 previous approach, all test cases defined by the complete scenario tree represented all possible stimulus and response behaviors 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 branch testing. 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 decision testing.

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 have a transition removed without destroying the covering property. Changing the goal to be a minimum covering, it can be shown that the problem of minimal test case generation can be reduced to the set covering problem and is therefore NP-complete.

4. A HEURISTIC APPROACH

Initially, no requirements will have been covered, so any available stimulus that causes a change in the model will help reach the goal. This suggests a greedy approach. This approach would expand all possible states for the most recent state added to the minimal scenario tree, *MST*, then add the state connected by the edge covering the most as yet uncovered requirements. The number of uncovered requirements, call this value the Requirements Covering Value or *RCV*, is the quality measure used to select the next state to be added to the scenario tree.

The greedy approach is not a complete solution. It is quite likely that the queue of potential states will at some point contain no entries that are associated with uncovered requirements (and all requirements are not yet covered). When this happens, a switch to a second strategy is needed, because the greedy approach would degenerate into a random selection process. What may occur is that a few requirements might only be covered by edges near or at the leaves of the tree. These transitions require that the system be taken through a long series of stimuli in order to get the system in a state that allows the needed transition to be taken.

From the requirements model, the transition associated with an elusive requirement can be analyzed to determine the enabling state (possibly with *don't care* values) for that transition. This enabling state can then be compared to each potential state in order to calculate a distance measure. The distance will be the sum of the differences for numeric state variables plus the number of enumerated state variables that do not match. The selection of the next state to expand will be made based on this measure, and the expansion will follow the most promising path so far.

During the distance based search phase of the algorithm, if the edge leading to the next state selected did not cover any additional requirements, it will not be added to the *MST*. This is because there is no guarantee that any test case 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 *MST*. All edges and states along this trace are then added to the *MST*.

Our heuristic algorithm uses the two phase approach just described: greedy search followed by a distance based search if needed. In the following paragraphs the 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 algorithm maintains graphs *ST* and *MST*, representing the scenario tree and minimal scenario tree, and sets *POT*, *COV*, and *RS*, representing the states (vertices) not yet part of the *MST*, the covered requirements, and the requirements remaining to be covered respectively. These are initialized as follows:

```
ST = graph with initial state vertex only
MST = Null graph
POT = { }; COV = { }; RS = { set of all requirements }
```

The top level structure of the algorithm is given below. The greedy search phase will continue until no further progress can be made. The potential state set is then cleansed of any states that already exist in the *MST* 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, if *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 $s_i \in POT$
  if $si \in$ States(MST) Delete $s_i$ from POT
if (RS is not empty and POT is not empty )
  Select $r_{target}$ from RS and determine it's enabling state $s_{en}$
else
  $r_{target}$ = NULL
while ($r_{target}$ != NULL ) {
  DistanceBasedSearch()
}
Output all rooted paths from MST
stop
```

In the description of the greedy search below, 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 *ST*. 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.

```
GreedySearch()
{
  if ( nextS $\notin$ States(MST) )
    ($S_{new}$, $E_{new}$) = Expand(nextS)
    Add ($S_{new}$, $E_{new}$) to ST and add $S_{new}$ to POT
  Add (nextS, nextE) to MST
  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
}
```

In the description of the distance based search below, the ComputeDist() function calculates the distances from each member of the given state set to the given enabling state and returns the state associated with the minimum value found. The

BackTrace() function add states and edges to *MST* by performing the backward trace as previously described.

```
DistanceBasedSearch()
{
  $s_{min}$ = ComputeDist($POT$, $s_{en}$)
  $S_{new}$,$E_{new}$ = Expand($s_{min}$)
  Add $S_{new}$,$E_{new}$ to $ST$ and add $S_{new}$ to $POT$
  for all $(s_i,e_i) \in (s_{new},e_{new})$
    if any member of Req($e_i$) $\in RS$
      BackTrace($s_i$)
      Move all Req($e_i$) from $RS$ to $COV$
  for all $s_i \in POT$
    if $s_i \in$ States($MST$) Delete $s_i$ from $POT$
  if ( $RS$ is empty or $POT$ is empty )
    $r_{target}$ = NULL
  elseif ( $r_{target} \notin RS$ )
    Select $r_{target}$ from $RS$ and determine it's enabling state $s_{en}$
}
```

5. TEST CASE GENERATION EXAMPLE

Consider the following example based on the control system described in [8] and simplified in [9]. We have modified the system slightly to more fully exercise our test generation algorithm. The example is of a controller for a safety injector of a reactor core. The system monitors pressure and adds coolant if the pressure drops below a given threshold. The operator may block the safety injection by toggling a *Block* switch and can reset the block by toggling a *Reset* switch. The system also receives a time reference input, *Tref*. If the pressure is below the threshold and the system is blocked, after the third *Tref* input is received, the system will automatically turn off the block and allow safety injection. Figure 2 shows a block

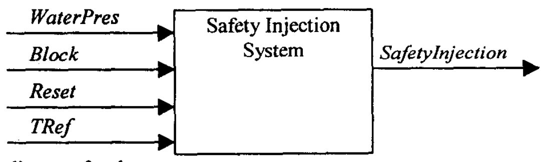

diagram for the system.

Figure 2. Safety Injection System

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

[R1] The system shall assert *SafetyInjection* when *WaterPres* falls below LOW as long as the system is not blocked.

[R2] The system shall be considered blocked in response to *Block* being asserted while *Reset* is not asserted, and shall remain blocked until either *Reset* is asserted or *WaterPres* crosses LOW in either direction.

[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 [9]. 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.

Table 1. Mode Transition Table for Pressure

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

Table 2. Event Table for Overridden

Mode	Events	
Permitted, TooLow	@T(*Block*=On) **[R2a]** WHEN *Reset*=Off	@T(Inmode) **[R2b]**
Permitted, TooLow	@T(False)	@T(*Reset*=On) **[R2c]**
TooLow	@T(False)	@T(TrefCnt=3) **[R4d]**
Overridden'	True	False

Table 3. Event Table for TRefCnt

Mode	Events	
TooLow	@T(*TRef*) WHEN **[R4a]** Overridden	@F(Inmode) **[R4b]**
TooLow	@T(False)	@T(Reset=On) **[R4c]**
TRefCnt'	TRefCnt + 1	0

Table 4. 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 requirements labels have been added to show the association to the original requirements. 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* $\in$ {Natural}, *SafetyInjection* $\in$ {On, Off}. The initial state of the system is (P, F, 0, Off). Note that *Overridden* is equivalent to *Blocked* in the textual requirements.

The execution of the algorithm described in the previous section is illustrated in Figure 3. The test stimuli for the *WaterPressure* input are limited to 50 and 150. Edges are labeled by *input/output* and covered requirements are listed in square brackets. The vertex numbering indicates the order in which the vertices are created and added as potential states. The edge number indicates the order in which edges are added to the scenario tree.

Initially the algorithm starts with the greedy approach, always selecting the potential state connected by the edge covering the most as yet uncovered requirements, and selecting at random in the case of a tie. The greedy portion of the

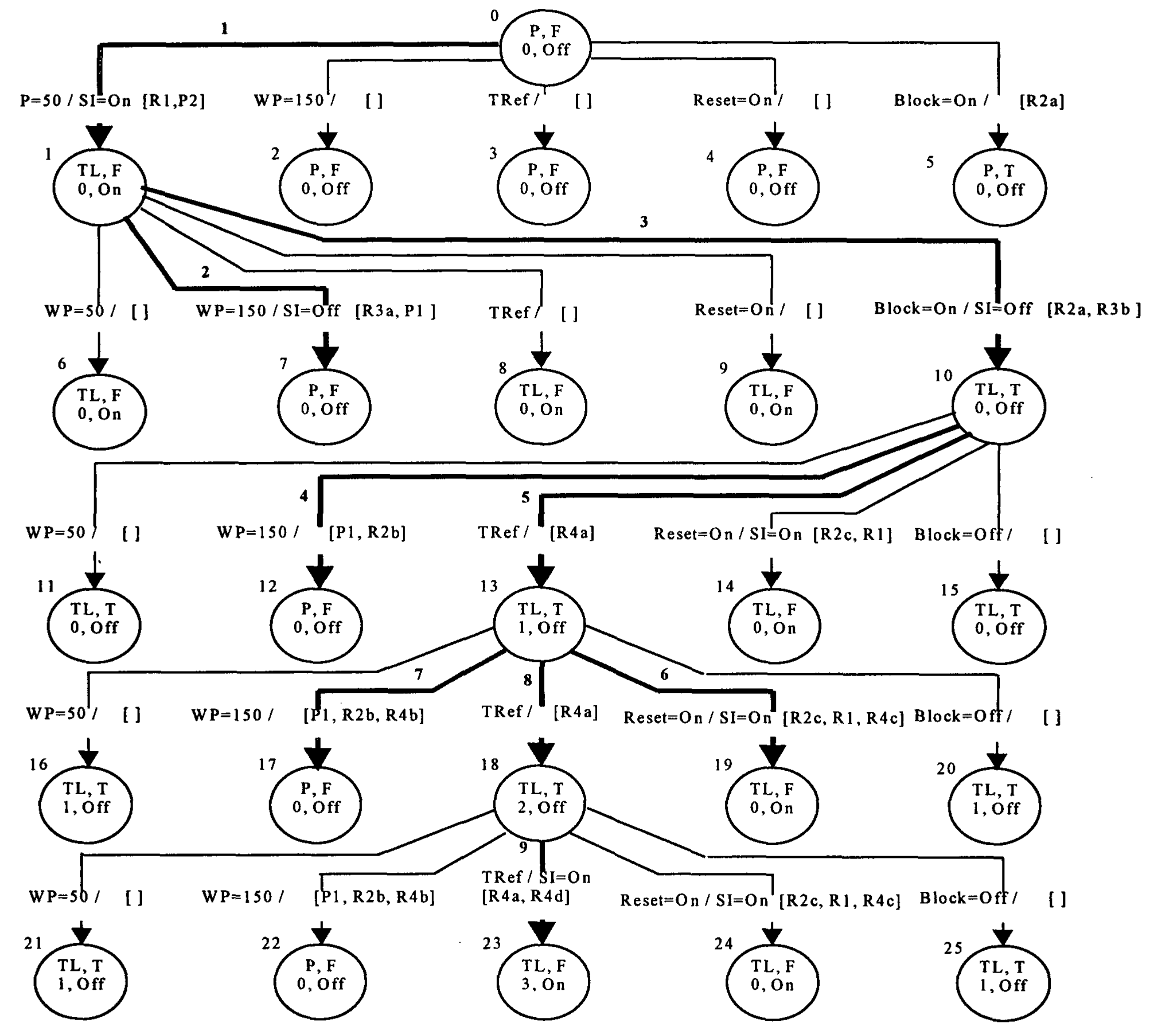

Figure 3. Scenario Tree for Safety Injection System

algorithm continues in this manner until the *MST* consists of states 0, 1, 7, 10, 12, 13, 17, and 19. One possible order for adding these states is indicated by the numerical edge labels. Note that states that are part of the scenario tree but have not been expanded are equivalent to states that are already part of the *MST*.

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 single remaining requirement is R4d. The state required to allow R4d is TL, X, 3, X where X is a don't care. At this stage the only expandable states are 5 and 18. The distance for state 5 is 4 (1 for the enumerated *Pressure* plus 3 for the difference in *TrefCnt*) and the distance for state 18 is 1 (for the difference in *TrefCnt*). Expanding state 18 reveals the desired transition leading to the addition of states 23 and 18 to the scenario tree.

The generated scenario tree contains the five scenarios listed below.

1 WP=50 / SI=On
WP=150 / SI=Off

2 WP=50 / SI=On
Block=On / SI=Off
WP=150 /

3 WP=50 / SI=On
Block=On / SI=Off
TRef /
WP=150 /

4 WP=50 / SI=On
Block=On / SI=Off
TRef /
Reset=On / SI=On

5 WP=50 / SI=On
Block=On / SI=Off
TRef /
TRef /
TRef / SI=On

6. CONCLUSIONS AND FUTURE WORK

An approach for automatic test case generation from system requirements has been presented. This approach is based upon a formulation into a covering problem The difficulty of this problem has been discussed and a heuristic algorithm to solve the problem has been given. The use of this algorithm has been illustrated using a small example.

Our approach is similar to the previous work done at GTE Laboratories [10]. Our work differs in that we are developing test cases for embedded hardware/software systems rather than purely software systems and in the way requirements are associated with test cases. In their approach, stimulus and responses are interactively associated with requirements as a test engineer requests test cases to be generated. In our approach, the association is determined when the requirements model is generated and the test cases are generated to ensure the covering as previously described.

The work of Hsia [4,11] is also relevant. Their domain is again, software systems. Scenarios are elicited from the customer and used to develop conceptual finite state machines. From these FSMs, the set of all possible scenarios are generated. The desire to not generate all possible scenarios is one of the primary motivations for our heuristic approach. Also, since Hsia has no explicit relation between requirements and scenario, the bases for selective generation of scenarios does not exist.

Blackburn [12] describes a method to generate test vectors from SCR style specifications by converting the SCR model into a formal logic description used by the T-VEC system [13]. The T-VEC system generates test vectors. Test vectors are singular entities applied at a single point in time. As a result, they do not represent a sequence of events and would not allow for the verification of temporal requirements.

The focus of our continued research will be focused on the evaluation of this method on various systems. In addition, a notation to capture the temporal requirements of systems will be determined and a method to apply and verify these requirements at the model and physical prototype level will be developed. Finally, it is desirable to investigate extending the requirements model beyond FSMs in order to apply this approach to a wider range of systems.

Acknowledgments

This work has been supported by the National Science Foundation under grant No. 9554561 "Hardware/Software Codesign for High Performance Systems." We would like to thank Tony Ewing and Stephan Schulz their help in the development of the heuristic algorithm.

7. References

[1] 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. 136-43, Nashville, TN, March 1999.

[2] Harel, D. "STATEMATE: A Working Environment for the Development of Complex Reactive Systems." *IEEE Transactions on Software Engineering*, 16(4), pp. 403-14, 1990.

[3] 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.

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

[5] Schulz, S., Rozenblit, J.W., Mrva, M. and Buchenrieder, K., "Model-Based Codesign," *IEEE Computer*, 32(8), 60-68, 1998.

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

[7] 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. 166-72, Nashville, TN, March 1999.

[8] Courtios, P.J., Parnas, D.L., "Documentation for Safety Critical Software," *Proceedings of the 15th International Conference on Software Engineering (ICSE'93)*, pp. 315-23, Baltimore, MD, 1993.

[9] 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. 231-61, July 1996.

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

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

[12] Blackburn, M.R., Busser, R.D., Fontain, J.S., "Automatic Generation of Test Vectors for SCR-Style Specifications," *Proceedings of the 12th Annual Conference on Computer Assurance (COMPASS'97)*, pp. 54-67, Gaithersburg, MD, June, 1997.

[13] Blackburn, M.R., Busser, R.D., "T-VEC: A Tool for Developing Critical Systems," *Eleventh International Conference on Computer Assurance (COMPASS'96)*, pp. 237-49, Gaithersburg, MD, June, 1996.