Synthesis of High-Level Requirements Models for Automatic Test Generation

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

Abstract

This paper describes research and development of techniques to support automatic generation of test cases for event-oriented, real-time embedded systems. A consistent suite of test scenarios can assure consistency at all levels of design activities. Although we have developed algorithms designed to generate test scenarios from state-based functional requirements model, their applicability is severely limited without a means to automatically translate the model functions into a form that can be readily integrated with the algorithms. A method and tool that extract the model of requirements and synthesize an equivalent high-level functional representation are presented. The tool, called Requirements Model Code Synthesizer, has been applied to a number of design cases, one of which is described in this paper.

1. Introduction

Our work is motivated by the need to improve the practice of embedded systems design. Hardware/Software Codesign [2] has been a recently active area of research that has emerged due to the same need. Codesign takes a systems approach by supporting an integrated hardware/software design environment. Model-based Codesign uses executable and realization independent simulation models to allow for early alternative design analysis and tradeoff studies. Iterative refinement is used in taking a proposed design from its model to its physical realization.

The Model-based Codesign process relies heavily on testing that is applied at all levels of the design activities. When a consistent suite of test cases are utilized, consistency across the design levels can be assured. To support the testing needs of Model-based Codesign, we have developed an automatic test scenario generation method. This method is based on a state-based model of the proposed system. The model is developed manually from the stated requirements for the system and is called the requirements model. An integral part of the test scenario generation method is the ability to automatically extract model functions and other model dependent information from the requirements model and convert it to a form suitable for use by the algorithms that perform scenario generation. The focus of this paper is to describe a method and tool that have been developed to provide this automation, and as a result, greatly improve the applicability of our test scenario generation method.

First, we provide an overview of our approach to automatic test scenario generation. In Section 3 we give a brief description of the semantics of the Software Cost Reduction (SCR) formalism [3], which was chosen as the state-based modeling formalism used to support our scenario generation method. Section 4 presents the approach taken in developing the Requirements Model Code Synthesis (RMCS) tool. A case study of an elevator controller is presented in Section 5 to illustrate the details of the RMCS tool. Section 6 concludes with a summary and suggests areas for future work.

2. Approach to test scenario generation

Test scenarios are designed to support the Model-based Codesign process. They are used across all levels of the design to ensure consistency between the different design representations (e.g., from functional and behavioral models at the virtual level to prototypes and final designs at the physical level). The suite of test scenarios should exercise (cover) all requirements so that the application of the test suite to a model or a system provides means to validate the design with respect to the system requirements.

Our approach to test scenario generation assumes that the system requirements will be stated in a natural language document. These requirements are then used to develop a state-based requirements model. The modeling formalism we presently employ is the SCR method developed at the Naval Research Laboratories. Requirements models are developed using the Software

Cost Reduction Toolset [4]. The requirements model defines the proper functionality of the proposed system and is assumed to accurately reflect the stated requirements.

A set of scenario generation algorithms has been developed that provide a controlled simulation of the requirements model in order to produce a rooted scenario tree [5]. The root of the tree is defined as the initialization state for the system. All paths from the root to pendent vertices define test scenarios. The algorithms are applied in a serial manner. The first algorithm produces a scenario tree that defines a set of scenarios, called the base scenarios, that exercise all requirements identified in the requirements model [6]. Because testing at the black box level is necessary to support testing within Model-based Codesign, the second algorithm determines which requirements can be verified at the black box level through application of the base scenarios. The third algorithm attempts to enhance the base scenarios for all requirements identified by the second algorithm as black box unverifiable. The fourth and final algorithm combines the enhancements and the base scenarios to produce a final suite of test scenarios. The details of the complete scenario generation process are given in [5].

Implementations of the scenario generation algorithms have been developed in the C programming language. These implementations are designed to compile and link with C code that is functionally equivalent to the requirements model. The algorithms control the requirements model functions through a standard interface. It is the job of the RMCS tool to provide the functionally equivalent model code, to generate all model dependent support code, and to provide the standard interface code. The RMCS tool is the key element that allows the scenario generation algorithms to be seamlessly applied to any requirements model developed in the SCR formalism.

3. Overview of the SCR formalism

In this section we present an overview of the SCR formalism [4] used in the requirements model. The SCR formalism is built upon the state-based mathematical model and provides two sets of expressions: condition expressions and event expressions. Condition expressions are treated in the same manner as C logical conditions. However, more logic is associated with event expressions and needs demonstration.

In the SCR formalism, an event occurs when the value of the expression changes from one state to another. The event expression *@T(A)* denotes the event when the value of expression A changes from *FALSE* to *TRUE*. If *A* is the condition's value in the old state and *A'* is its value in the new state then,

$$@T(A) = A' \text{ AND NOT } A. \quad (1)$$

Figure 1 illustrates this relationship graphically[1].

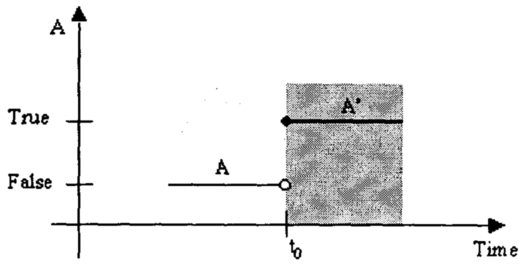

Figure 1. Definition of the SCR construct *@T(A)*. When the value of expression *A* changes from *FALSE* to *TRUE* at t_0, we say that event A occurred at t_0.

The equivalent C expression for the *@T(A)* statement is

$$if(\, A' \;\&\&\; !A \,)\,\{\,...\,\}. \quad (2)$$

Similarly, the event expression *@F(A)* denotes the event when the value of expression *A* changes from *TRUE* to *FALSE*. The expression is defined as

$$@F(A) = A \text{ AND NOT } A' \quad (3)$$

and its equivalent C expression is

$$if(\, A \;\&\&\; !A' \,)\,\{\,...\,\}. \quad (4)$$

Table 1 summarizes the SCR event constructs, their mathematical definitions, and their equivalent C expressions.

We note in the last row of Table 1 that *B* is a condition expression that is a part of the conditional event expression. From these basic constructs and the *AND/OR/NOT* keywords, multiple events and conditions can be grouped to form more complex expressions. Nevertheless, the RMCS tool will still use the logic behind the fundamental constructs to decode these complex expressions.

The three types of tables used in the SCR formalism are the mode-transition, the event, and the condition

[1] This figure has been reproduced from the *SCR Toolset: The User Guide*, courtesy of the Naval Research Labs.

Table 1. SCR event constructs

SCR Construct	Mathematical Definition	Equivalent C Expression
@T(A)	*A' AND NOT A*	`if ( A' && !A ) { ... }`
@F(A)	*A AND NOT A'*	`if ( A && !A' ) { ... }`
@C(A)	$A' \neq A$	`if ( A' != A ) { ... }`
@A	$\exists X_{monitored} (X'_{monitored} \neq X_{monitored})$	`for ( i = 0; i < num_monitored; i++ )` `if ( X'`$_{monitored}$`[i] != X`$_{monitored}$`[i] )` `monitored_change = TRUE;` `if ( monitored_change ) { ... }`
@T(A) WHEN B	*A' AND NOT A AND B*	`if ( A' && !A && B ) { ... }`

tables. In addition, there are four types of variables: modeclass, monitored, term, and controlled. Monitored and controlled variables are the inputs and the outputs of the system, respectively, while term variables are internal to the system. Modeclass variables capture the system history because the behavior of a system can very well depend upon what happened in the past. In using the SCR formalism, the modeler must assign a mode-transition table to each of the modeclass variables and either an event or a condition table to each of the term and controlled variables.

4. Tool development approach

The methodology applied in developing a suitable tool to synthesize equivalent functional model code from the SCR requirements model is based upon the principles of compiler design. (We feel that if industry is to adopt our approach, we must build upon existing tools to help improve efficiency. Consequently, we extensively used the UNIX tools lex and yacc and the Solaris Workshop development environment in our tool development process.) The inputs, the outputs, and the architecture of the Requirements Model Code Synthesis tool are shown in Figure 2.

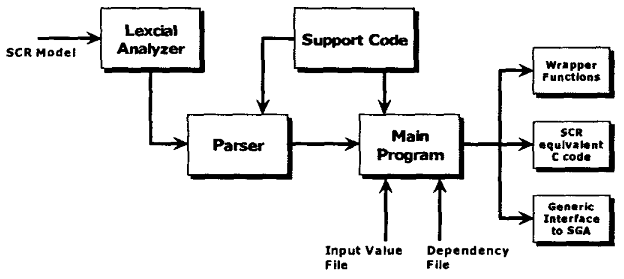

Figure 2. The inputs, the outputs, and the architecture of the Requirements Model Code Synthesis (RMCS) tool.

4.1 Tool inputs

The RMCS tool requires three inputs: the SCR requirements model, the input value, and the dependency files. The SCR requirements model file contains the model of the system for which test scenarios must be generated. The input value file contains the possible values of all input variables. These values define the inputs that the scenario generation algorithms are allowed to apply to the requirements model during state expansion. The dependency file enumerates all the model variables and indicates the dependencies present between variables in the model. The scenario generation algorithms use the dependency information during simulation of the requirements model.

4.2 Tool architecture

The design of the RMCS tool comprises three major components. As shown in Figure 2, the components are a lexical analyzer, a parser, and the support code. The main program calls the parser which parses the SCR requirements model-file by developing a parse tree and a symbol table that are used to generate the SCR equivalent C code. In order to parse the file, the parser requests the help of a lexical analyzer. The equivalent functional model code is generated by the support code after parsing has been accomplished.

The lexical analyzer (lexer) is written with lex to analyze the text file containing the SCR requirements model. In a lexer, the user defines a series of patterns that must be identified. The lexer searches for these patterns in the model and breaks the contents that are matched into a series of tokens. The parser will process these tokens further. An error is reported and further processing is halted if the lexer detects tokens that do not conform to any of the defined patterns.

The parser, developed using yacc, initiates a call to the lexer asking for tokens. It logically groups the supplied tokens in accordance with the syntax rules defined by the user. In our case, the syntax of the SCR formalism was defined in the *SCR Toolset: The User Guide*. The support code associated with each syntax rule is executed when that rule is matched. In our case, the primary purpose of the parser is to generate a parse tree and a symbol table. An error is reported if any of the syntax rules are violated and further processing is terminated.

The parser and the main program utilize the supporting C code that was developed in the Solaris Workshop development environment. The parser uses the support code to perform the necessary actions required when a certain syntax rule has been identified. On the other hand, the main program uses the support code to process the parse tree generated by the parser and to generate the appropriate outputs.

We developed the RMCS tool in the Solaris Workshop environment primarily because it provides excellent tool development features such as debugging and maintenance. Although lex and yacc do not provide good error recovery during lexical analysis and parsing, we employed their use for two reasons. First, a lexer written in lex and a parser written in yacc are much more compact than both written in C. In addition, the tools provide the capability of automatically generating equivalent C source code that is ready for compilation with the support code and the main program. Second, we make the assumption that the model has passed all model checking tests provided by the SCR Toolset (e.g., syntax, variable, type, name uniqueness, disjointness, etc.). Thus, there is no need to repeat these tests in the RMCS tool.

4.3 Tool outputs

There are three outputs generated by the RMCS tool. Each table in the model is mapped onto an equivalent C function. These functions capture the same logic as that encoded in the model. But because these functions are model dependent, wrapper functions are synthesized for each function to help provide a generic interface that allows communication between the scenario generation algorithms and the model functions during test scenario synthesis. Finally, a generic interface to the scenario generation algorithms is generated which defines all data types and structures, determines state changes, handles data transfer, and provides input and output.

5. Case study: An elevator controller

We illustrate the application of the RMCS tool in test scenario generation with a case study. We defined the system requirements of an elevator controller in a text-based document from which we developed an SCR model. Partial tables excerpted from the actual model and the equivalent functional code assembled by the RMCS tool are given. Finally, we present a description of one of the generated test scenarios.

5.1 SCR elevator model

An elevator controller controls the operation of an elevator in a three-story building. The main floor has both up and down buttons while the ground and top floors have only up or down buttons, respectively. There are three inputs and three outputs in the SCR elevator requirements model as shown in Figure 3.

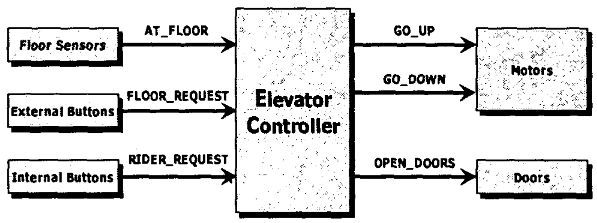

Figure 3. The inputs and the outputs of the SCR elevator requirements model.

The input and output variables are *At_Floor*, *Floor_Request*, *Rider_Request* and *Go_Up*, *Go_Down*, *Open_Doors*, respectively. The elevator uses *At_Floor* to keep track of the current floor position of the elevator. The variables *Floor_Request* and *Rider_Request* capture the floor selection made by the rider outside and inside the elevator, respectively. The outputs *Go_Up* and *Go_Down* control the elevator's direction of movement while *Open_Doors* controls the opening and closing of the elevator doors.

5.2 System Requirements

The main requirements of the elevator controller can be summarized as follows:

- The elevator must respond within 0.5 seconds of when a service request (internal or external) is made. A valid response is asserting *Go_Up* or *Go_Down* to command elevator movement. For simplicity, a sensor to detect door obstructions has been omitted.

- If the elevator is servicing a request, a new request that can be serviced without changing the direction of motion must be completed prior to completing the present request.

- Once a service request has been completed, the next pending request selected is the request at the closest floor in the same direction as last motion. If there are no such requests, then the request at the closest floor in the opposite direction of the last motion is serviced.

- When the elevator arrives at a floor to service a request the doors shall be opened by asserting *Open_Doors* within 0.5 seconds.

- A service request is completed when the elevator has arrived at the proper floor and the doors have remained open for 5 seconds. In the absence of any service requests, the elevator shall remain at the present floor.

5.3 Examples of SCR tables and generated functional code

There were two mode-transition, nine event, and one condition tables in our SCR elevator controller model. Table 2 shows a portion of the mode-transition table for the modelcass variable *ControlMode*. The first column indicates the old value of *ControlMode*. The last column indicates the new value of *ControlMode* should the event in the second column occur. For example, if *ControlMode* was in the *Waiting* mode and *Floor_1_Request* becomes true when *Cur_Floor* is 1.0, then *ControlMode* should transition into the *Service* mode.

The equivalent model code generated by the RMCS tool for *ControlMode* is shown below. In keeping with the SCR formalism, prime notation is used to denote the new state of the variable. For the sake of clarity and brevity, some of the code has been slightly modified.

```
if( ControlMode == Waiting )
 if( Floor_1_Request' && !Floor_1_Request &&
 Cur_Floor == 1.0 )
   ControlMode' == Service;

if( ControlMode == Go_Down )
 if( At_Floor' && !At_Floor && Cur_Floor == 2.5
 && Floor_2_Down_Request)
   ControlMode' == Service;
```

A portion of the event table for the term variable *Cur_Floor* is shown in Table 3. The first column indicates that *Cur_Floor* is dependent upon the modeclass variable *ControlMode*. Hence, its value can change only when *ControlMode* is in one of the modes and the indicated event(s) occur. For example, if *ControlMode* is in the *Go_Down* mode and *At_Floor* becomes true when *Cur_Floor* equals 1.5, then the new value of *Cur_Floor* is 1.0. However, if *ControlMode* is in the *Go_Up* mode, *Cur_Floor* will never equal 1.0 because the *NEVER* event is associated with this assignment. Thus, it does not need to be captured in the generated model code.

The model code corresponding to the event table for *Cur_Floor* is shown below.

```
if( ControlMode == Go_Up )
 if( !At_Floor' && At_Floor && Cur_Floor == 1.0 )
   Cur_Floor' = 1.5;

if( ControlMode == Go_Down ) {
 if( At_Floor' && !At_Floor && Cur_Floor == 1.5 )
   Cur_Floor' = 1.0;
 else if( !At_Floor' && At_Floor && Cur_Floor ==
 2.0 )
   Cur_Floor' = 1.5;
}
```

A condition table is translated in a similar manner to an event table. The only difference is that, instead of events, the specified conditions must be true for state transitions to occur. In addition, the prime notation is not utilized here because condition tables indicate the variable's value in any particular state. The condition table for the controlled variable *Open_Doors* is shown in Table 4. For example, if *ControlMode* is in *Waiting*, *Go_Up*, or *Go_Down* mode, then the value of *Open_Doors* is *FALSE*.

The functional code for the *Open_Doors* condition table is shown below.

```
if( ControlMode == Service ) {
 if( Open_Doors )
   Open_Doors = TRUE;
 else if( !Open_Doors )
   Open_Doors = FALSE;
}
```

Table 2. Partial mode–transition table for *ControlMode*

FROM	EVENT	TO
Waiting	*@T(Floor_1_Request) WHEN (Cur_Floor = 1.0)*	*Service*
Go_Down	*@T(At_Floor) WHEN (Cur_Floor = 2.5 AND Floor_2_Down_Request)*	*Service*

Table 3. Partial event table for *Cur_Floor*

ControlMode		EVENT	
MODES	*Go_Up*	*NEVER*	*@F(At_Floor) WHEN (Cur_Floor = 1.0)*
	Go_Down	*@T(At_Floor) WHEN (Cur_Floor = 1.5)*	*@F(At_Floor) WHEN (Cur_Floor = 2.0)*
ASSIGNMENTS		*1.0*	*1.5*

Table 4. Condition table for *Open_Doors*

ControlMode		CONDITIONS	
MODES	*Service*	*True*	*False*
	Waiting, Go_Up, Go_Down	*False*	*True*
ASSIGNMENTS		*True*	*False*

```
if ( ControlMode == Waiting || ControlMode ==
    Go_Up || ControlMode == Go_Down ) {
  if( !Open_Doors )
   Open_Doors = TRUE;
  else if( Open_Doors )
   Open_Doors = FALSE;
}
```

5.4 Generated scenarios

The equivalent, functional model code generated by the RMCS tool compiles and links with the model-independent scenario generation algorithms and its associated support libraries as shown in Figure 4. This produces a self-contained executable scenario generator for the given requirements model.

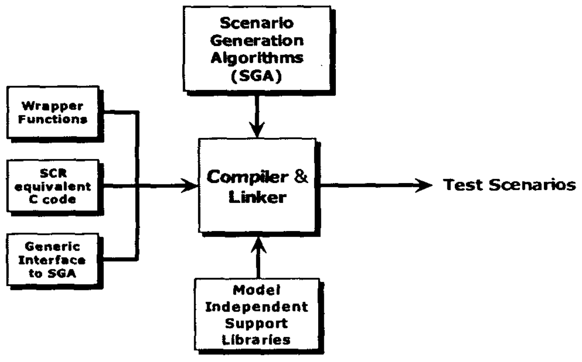

Figure 4. The outputs of the RMCS tool compile with the model-independent scenario generation algorithms and support libraries to generate test scenarios.

The scenario generation algorithms generated seventy-three unique system states for the elevator model and produced twenty-two test scenarios. One of the test scenarios is shown in Figure 5. The format of the test scenarios is *state input/output*.

```
SCENARIO 19
--------------
40   FLOOR_1_INT_BUTTON_PRESS /
     OPEN_DOORS = TRUE
 48  FLOOR_3_DOWN_EXT_BUTTON_PRESS /
 62  TimeAdvance = 5.010 /
     OPEN_DOORS = FALSE GO_UP = TRUE
 73
```

Figure 5. A sample test scenario generated by the scenario generation algorithms.

The specified initial state for the elevator controller is that the elevator is on the first floor with the doors closed and no pending requests. From this state, the first step in the scenario is for a passenger (this particular scenario also assumes that a passenger was in the elevator) to request to go to the first floor. Since the elevator is currently at the first floor, the doors should be opened. The second step shows that while the doors are open, someone on the third floor requests to go down. The third step shows the advancement of time past the fixed open door delay which should result in the doors being closed and the elevator beginning its ascent to service the request on the third floor. This scenario stops at this point, rather than the logical conclusion of servicing the third floor request, because any requirements that could have been tested by continuing have already been covered by other scenarios.

6. Conclusions and future work

In this paper, we have presented the development of the RMCS tool to aid the test scenario generation algorithms. By automating the translation of the high-level requirements model, our scenario generation algorithms can be applied to any requirements model developed in the SCR formalism. This eliminates the need to write test cases manually and expedites the design process. In addition, industry can benefit immensely from this structured framework of test generation for system validation, which will help reduce design cycles and costs. Our future work in this area of embedded systems design is geared towards combining the generated test scenarios with temporal (performance) requirements to generate C/ATLAS (Common Abbreviated Test Language for All Systems) [7] test programs that will be used in a virtual or real-time testing environment for system validation.

7. References

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

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

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

[4] Heitmeyer C.L., Kirby, J., and Labaw B.G., "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.

[5] Cunning, S.J. "Automating Test Generation for Discrete Event Oriented Real-Time Embedded Systems." Ph.D. Dissertation for the Department of Electrical & Computer Engineering, The University of Arizona, Fall 2000.

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

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