Test Scenario Generation from a Structured Requirements Specification

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

Abstract

A brief overview of the requirements engineering, its history, and state of practice are given. A semi-formal method to structure the behavioral requirements for real-time embedded systems is presented. This method is based on a set of forms that contain both informal text-based descriptions and formally defined language constructs. After documentation of requirements into these forms, an algorithm to automatically generate event scenarios is presented. This algorithm extracts the needed information from the requirements forms and produces a set of scenarios that can be used to test transaction oriented systems. A design example is presented that is used to illustrate the process of converting text based requirements onto the structured requirements form and to illustrate the operation of the scenario generation algorithm

.

1. Introduction

Much work in the area of requirements engineering has been done over the past twenty years. Many languages have been developed and implemented [1,2,3,4]. Many methods have been proposed [5,6,7,8,9,10] and White provides a comparative analysis of eight such methods in [11]. Throughout these works, the problem statement, which has remained essentially unchanged over the years, is that incomplete, ambiguous, incompatible, and incomprehensible requirements lead to poor designs. The conclusions have also been consistent. The use of a structured requirements language, usually with tool support and within a requirements solicitation and documentation process leads to early detection of problems and misunderstandings. The resolution of which leads to better designs.

The question raised by personal experience and echoed in the literature [12,13] is "Why aren't these languages and methods in wide spread use today?" Many valid reasons are given, and it is certainly true that management and engineering both need to believe in the necessity of the requirements engineering effort and in the benefits of the end product, the requirements specification. Another factor that may be hindering the acceptance of worthy improvements is that they present too large of a leap for many organizations to take. In other words, improvements are developed without paying enough attention to the state of practice, and then it is hoped that they will catch on. We would argue that a better approach would be to build on current practice, i.e. languages, tools, and methods in use today. This will more easily guide industrial organizations toward improved requirements engineering and the resulting benefits.

2. Rationale for a structured requirements specification

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

At the other extreme, the requirements may be converted into a language based on formal logic. The formal description of requirements allows for the use of theorem proving methods in order to prove the correctness and consistency of the given set of requirements. A selected summary such specification languages can be found in [14]. These methods, however, are not widely used in practice due to difficulty in the conversion process and the fact that most designers have not been trained in these formal methods.

As a middle of the road approach, we propose the use of a Structured Requirements Specification (SRS). The structure as presented here is intended for use in the

domain of real-time embedded systems design. This approach will provide the designer with a structured set of forms that will be used to convert from the native language requirements to semi-formal requirements. These forms will allow the designer to more easily identify requirements deficiencies.

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

In addition to the benefits mentioned above, properly structured requirements can be used to aid in other aspects of the design process. The SRS as described in this paper is intended to be used as the first step in our model-based codesign methodology [15,16]. The process is summarized in Figure 1. This design process relies heavily on modeling and simulation in order to verify the functionality of the design prior to implementation. The modeling in our process begins with a functional decomposition followed by behavioral modeling to produce an executable model suitable for simulation. As a benefit of using the SRS forms, a top level functional decomposition of the system can be easily extracted.

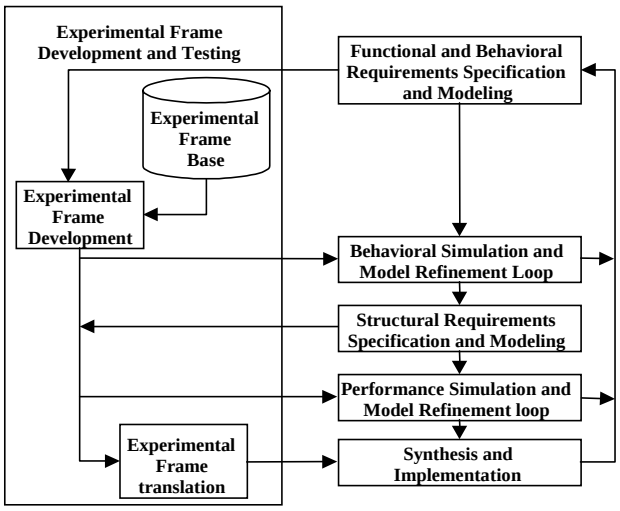

Figure 1. Model-based Codesign

With the use of high level simulation comes the need for test cases. The need for test cases will also exist for design processes that do not include high level simulation, since most will include simulation at lower levels and there will always be a need to test the physical prototypes. The use of the SRS will help to automate the activity of test case generation. From analysis of the information contained in the SRS, event scenarios based upon the allowed input and output relationships can be synthesized. Test cases for the behavioral models in our design process are used to define *experimental frames*.

The key concept of this aspect is *experimental frame*, i.e., the specification of circumstances under which a model (or a real system) is observed and experimented with. An experimental frame reflects modeling objectives since: a) it subjects a model to input stimuli (which represent potential interventions into the model's operation); b) it observes the model's reactions to the input stimuli and collects the data about such reactions (output data); and 3) it controls the experimentation by placing relevant constraints on values of the designated model state variables and by monitoring these constraints. Experimental frames are given concrete form; employing the concepts of automata theory and the DEVS (Discrete EVent System specification) formalism, Zeigler [17] defines a *generator* which produces the input segments sent to a model, an *acceptor*, i.e., a device that continually tests the run control segments for satisfaction of the given constraints, and a *transducer* which collects the input/output data and computes the summary mappings.

The set of *experimental frames*, whether coupled to an executable model or translated into a test environment to test a physical realization of the design, will be used at all levels of the model-based codesign process, as shown in Figure 1.

Automatic generation of test scenarios had received some attention in earlier research [18,19]. However, recent research on this topic is scarce. Industrial experiences has shown that ad hoc and code-based methods are still in practice. Full automation of all types of test cases is most likely a long way off, but the need for practical improvements to this area certainly exists. The primary goal of this paper is to present a systematic method that may be used to automate the generation of a certain class of test cases.

3. Description of SRS forms

The SRS is composed of multiple forms in order to cover different requirement aspects. The formulation of the SRS forms emphasizes the separation of interface and functional requirements. Temporal requirements are specified in terms of relationships at the system interfaces. Functional requirements define data and state transformations for the system.

Table 1 lists the fields of the I/O Requirements From. The Name field will specify a unique identifier for the I/O item. The Source field defines a link back to the

source of the requirement. The Direction field will specify one of the three direction types. The Physical field will be a text based description of the physical requirements for the particular interface. The Format field will define the logical format of the interface. If the format is specified to be Enumerated or Analog then a definition will follow. If the format is specified to be Bus, then the valid messages and associated direction for each must be specified. The next field is used to define the availability of inputs or the update requirements for outputs. It will be defined as one of the four listed types. Continuous is used only for inputs since the assumed design domain is that of digital systems. If defined to be Periodic, a nominal, minimum, and maximum rate must be specified. If Event or State, the minimum/maximum separation and relationships to other inputs or outputs must be specified as appropriate (for State types, the change of state is treated as an event). The Initial Value field is used to specify the required initial value for outputs of type State.

Table 1. I/O Requirements Form

Field	Description
Name	Symbolic
Source	Requirement ID
Direction	Input | Output | Bi
Physical	Physical characteristics
Format	Enumerated | Analog | Bus
Availability /	Continuous | Periodic | Event | State
Initial value	Format dependent

Table 2 lists the fields of the Functional Requirements Form. The Name and Source field are defined in a similar manner as on the I/O Requirements Form. The Inputs field will be used to specify a list of all inputs needed by the given function. The Output field will specify all outputs generated by the given function. The Enabled field is used to specify when the function is enabled by expressing a condition expression. The Transformation Definition field will be used to specify the transformation performed by the named function.

One of the integral steps within our model-based codesign methodology is the creation of an executable behavioral model of the proposed system. This model may be specified in any number of simulatable languages (e.g. Statecharts [20] or DEVS [21]. The functional definitions may then be extracted from the model and a pointer placed in the Transformation Definition field. This will eliminate the need for the nearly duplicate effort of developing a functional model within the SRS forms. In this way, modules of the behavioral model will form a part of the requirements specification.

Table 2. Functional Requirement Form

Field	Description
Name	Symbolic
Source	Link to source of the requirement
Inputs	List of Names
Outputs	List of Names
Enabled	After | Before | While | Always | Every
Transformation	Structured language

In the case where the structured requirements forms are used outside of model-based codesign, the Transformation Definition may be specified in the form of a structured language. The structured language must be able to represent conditional (e.g. IF, THEN, etc.) and data manipulation expressions (e.g. value assignment), but does not need to represent temporal relations. For simplicity, the functional definitions presented in this paper will be given in the C programming language.

During specification of the Transformation Definition, it will almost certainly be necessary to specify internal state variables. These state variables are needed to "remember" I/O events that don't otherwise cause an observable and persistent change at the system interface. These variables will be documented in a separate form called the Internal State Variables form. On this form the name, type, format, and initial value for each item will be specified. It should be noted that these variables are used solely in the requirements specification and do not impose a design choice.

4. Requirements definition and test generation method

It is assumed that most initial requirements will be in native language text format. We shall refer to these as the base requirements document. This document will then be used to fill out the SRS forms. This process will force those performing the transformation to consider aspects that may have previously been overlooked. This will occur if a field within a form cannot be confidently defined or conflicts with a related field. This situation exposes the need to return to the customer in order to come to an agreement on the missing or inconsistent requirement.

When the requirements definition team is satisfied with the I/O and Functional Requirements forms, event scenarios may be generated. In order to generate event scenarios, the behavior of the external systems and the behavior of the system being designed must be known. The former is defined through the interfaces through which the system must communicate (at least to the extent needed for transaction oriented systems) and is captured

on the I/O Requirements form. The latter is defined on the Functional Requirements form.

The process to get from text-based requirements to a generated set of test scenarios is depicted in Figure 2. Necessary information is extracted from the SRS forms and combined with the Scenario Generation Algorithm (SGA) to produce an executable model. The executable will be run to produce the scenario tree. Finally the scenario tree will be processed to output the list of event scenarios annotated with any timing requirements.

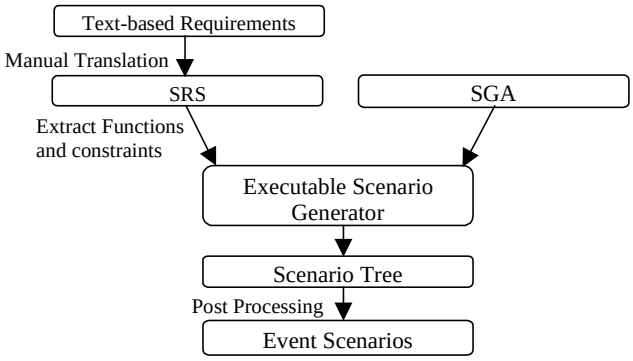

Figure 2. Process from Requirements to Scenarios

In essence, event scenario generation is an adaptive simulation of the partially executable specification defined in the SRS forms. The requirements specification is processed into an executable model. This model contains the functions as specified on the Functional Requirements form. The control structure for the system is embedded in the adaptive SGA. The SGA will evaluate the functions based on the constraints specified in the Enabled fields and will apply inputs to the system based upon the constraints specified on the I/O requirements Form. A high level representation of this algorithm is given below.

1. Create list of state variables that define the system state.
2. For each system function, create an enable expression.
3. Create an initial state vertex (root) with an empty Updated List and add it as the only member of the Active List.
4. Remove a vertex, *va*, from the Active List
4.1. For each item in the Updated List (if any) determine enabled functions and create new potential state vertices by evaluating these functions. Connect the new vertex to *va* with an edge labeled with the item from the Updated List and any system output generated by the evaluated functions.
4.2. If the Update List of *va* is empty
4.2.1. For each system input that applies to the state defined by *va*, determine enabled functions and create new potential state vertices by evaluating these functions. Connect the new vertex to *va* with an edge labeled with the input and any output generated by the evaluated functions.
5. If new vertices where created by step 4
5.1. For each vertex, *vn*, added in step 4
If *vn* defines a state not previously reached
Add *vn* to the Active List.
Else
5.2. If the Update List for *va* is not empty
Empty the Update List for *va* and return it to the Active List
6. Repeat steps 4 and 5 until the Active List is empty.

System state variables include all persistent system outputs (i.e. type is not Event) and all variables listed on the Internal State Variables form with a type of State. Initial values are extracted from the appropriate requirements form.

Enabling expressions are derived from the Enabled field of the Functional Requirements form for each system function. These expressions are needed to allow the scenario generation algorithm to easily determine when a particular function should be evaluated.

The scenario generation algorithm builds a scenario tree in a breadth first manner. The nodes represent system state and the edges are labeled with input/output list pairs. New nodes are created by applying external inputs to the system. System state changes are propagated until the system becomes stable (i.e. an external input or advancement of time is needed to produce a further change in state). Branches in the tree are expanded until all pendent nodes represent states that have previously been reached.

It should be noted that the full state space of the requirements specification will not be expanded. The states derived will be restricted in two ways. First by the specified availability and relations of the system inputs and secondly through the use of the explicit test inputs which restrict the scenario generation algorithm to a domain subset for selected large domain inputs, including time.

5. Design example using the SRS

To illustrate the use of the SRS forms, a temperature control system (TCS) will be used as an example. A small set of selected requirements for the TCS will be given in text form. The corresponding entries in the SRS forms will then be filled out by extracting the necessary information. Finally, execution of the SGA will be illustrated using this example.

Below are the selected text based requirements for the TCS. The requirement numbers have been added in

order to facilitate requirements traceability. References to system inputs, outputs, I/O values, and implied system variables are displayed in italics.

Table 3. Text based Requirements for the TCS

R1: The system shall have an input called *Control Disable*, which is a TTL compatible discrete signal.

R2: The system shall have an input/output bus called *Message Bus*, which shall be RS232 compatible.

R6: The system shall have an output called *Status*, which shall be a pair of TTL compatible discrete signals.

R8: The *Set Temperature* shall be set through the following procedure:

1. If the *Status* output indicates *Controlling*, *Control Disable* shall be provided to the TCS. The TCS shall respond by setting *Status* output to *Idle* within 10ms.
2. At least 10ms after the TCS has indicated Idle, a *Set Temperature* message may be sent to the TCS.
3. Within 10ms of receiving the *Set Temperature* message, the TCS shall respond with a *Temperature Valid* message if the received temperature is valid, otherwise the *Status* output shall be set to *Fault*.
4. At least 1ms after the TCS has responded with a *Temperature Valid* message, the *Control Disable* signal may be removed.

R9: A valid set temperature for the TCS is between 0°C and 110°C inclusive.

R10: The TCS shall begin controlling the *Temperature* within 1ms of the removal of *Control Disable* and shall indicate this by setting the *Status* output to *Controlling*.

R14: If a *Control Enable* signal is received when the TCS has not yet received a valid *Set Temperature*, or the most recently received *Set Temperature* message was invalid, the *Control Enable* signal shall be ignored and the system shall remain *Idle*.

5.1 Translation to SRS Forms

The translation process should begin with the I/O requirements form in order to define the system interfaces. The I/O Requirements form entries corresponding to the given text based requirements are listed in Table 4.

Table 4. TCS I/O Requirements

Name:	ControlDisable
Source:	R1
Direction:	Input
Physical:	TTL compatible
Format:	Enumerated - 0 = OFF, 1 = ON
Availability:	Event – 0: at least 1ms after ValidTemp; 1 at least 100ms after *Control Disable* = 0
Name:	Status[1:0]
Source:	R6
Direction:	Output
Physical:	TTL compatible
Format:	Enumerated - 00 = IDLE, 01 = CONTROLLING, 10 = FAULT
Update:	State - at most 10 ms after ControlDisable or ValidTemp
Initial value:	IDLE
Name:	MessageBus
Source:	R2
Direction:	Bi
Physical:	RS232 compatible
Format:	Bus -
	SetTempMsg - Input ID: 2 bits: fixed: 00; Temp: 16 bits: analog: 2's compliment - LSB=0.01°C
	ValidTempMsg – Output ID: 2 bits: fixed: 01
Avail./ Update:	SetTempMsg: Event - at least 10ms after Status = IDLE
	ValidTempMsg: Event - at most 10 ms after (SetTempMsg and ValidTemp = TRUE)

System functions are defined by extracting the behavior described in the base requirements. The Functional Requirements form entries corresponding to the given text based requirements are listed in Table 5.

Table 5. TCS Functional Requirements

Name:	StatusControl
Source:	R8, R10, R14
Inputs:	ControlDisable, ValidTemp
Outputs:	Status
Enabled:	After ControlDisable or ValidTemp

Trans. Def.:	if (Status == IDLE) {
	  if (ControlDisable == OFF) Status = CONTROLLING;
	  if (ValidTemp == FALSE) Status = FAULT; }
	else if (Status == CONTROLLING)
	  if (ControlDisable == ON) Status = IDLE;
	else if (Status == FAULT)
	  if (ValidTemp == TRUE) Status = IDLE;

Name:	ProcessSetTemp
Source:	R9
Inputs:	Status, MessageBus
Outputs:	ValidTemp, MessageBus, SetTemp
Enabled:	After MessageBus while Status = IDLE

Trans. Def.:	if (MessageBus.ID == 0)
	  SetTemp = MessageBus.Temp * 0.01;
	if (SetTemp >= 0 && SetTemp <= 110) {
	  MessageBus = ValidTempMsg;
	  ValidTemp == TRUE; }
	else
	  ValidTemp = FALSE;

The Internal State Variables form entries corresponding to the given text based requirements are listed in Table 6.

Table 6. TCS Internal State Variables

Name:	SetTemp
Type:	State
Format:	Real : LSB = 0.01°C
Initial value:	0.0°C
Name:	ValidTemp
Type:	State
Format:	Enumerated : 0 = FALSE, 1 = TRUE
Initial value:	FALSE

5.2 Scenario Generation

Scenario Generation proceeds as described in section 4. The state of the TCS (for the given subset of requirements) is defined by the set {ControlDisable, Status, SetTemp, ValidTemp}. The initial state is defined by {ON, IDLE, 0, FALSE}.

A potential problem for the SGA are inputs with a large domain of potential values. Attempting to apply a large set of inputs that vary only slightly will generally lead to a large number of system states that do not vary in a useful manner. This will cause the SGA to create many potential states only to find that most have been previously reached. In order to restrict the SGA, a set of test values will be defined for large domain inputs. This set will be a subset of the possible values for the given input. The test values will be selected at and near values that produce alternate decisions in the system function definitions. For the TCS example, *SetTempMsg* is one such input. One set of test values for this input could be:

SetTempMsg: 1=-50°C, 2=0°C, 3=110°C, 4=90°C

The enable expressions in the TCS example are:

StatusControl:	Available(ControlDisable) or Updated(ValidTemp)
ProcessSetTemp:	Available(SetTempMsg) and Status = IDLE

The use of Updated() indicates that the given item has been assigned a new value relative to the state in the parent node of the scenario tree.

With the preliminaries complete, scenario generation can begin. Although the generation algorithm generates all valid scenarios that lead to a unique system state, the following discussion will trace a single path through the scenario tree. Initially the Availability fields of the I/O requirements form are checked against the initial state to determine that *SetTempMsg* is a valid input to the system. Tracing along the *SetTempMsg*_3 path, the function *ProcessSetTemp* is evaluated. Evaluation produces the output event *ValidTempMsg* and produces a new system state with *SetTemp* = 110°C and *ValidTemp* = TRUE. For this new state the simulation time is advanced to 0.01s since the *SetTempMsg* is specified to occur at least 10ms after *Status* = IDLE and the Updated list is set to include *SetTemp* and *ValidTemp*.

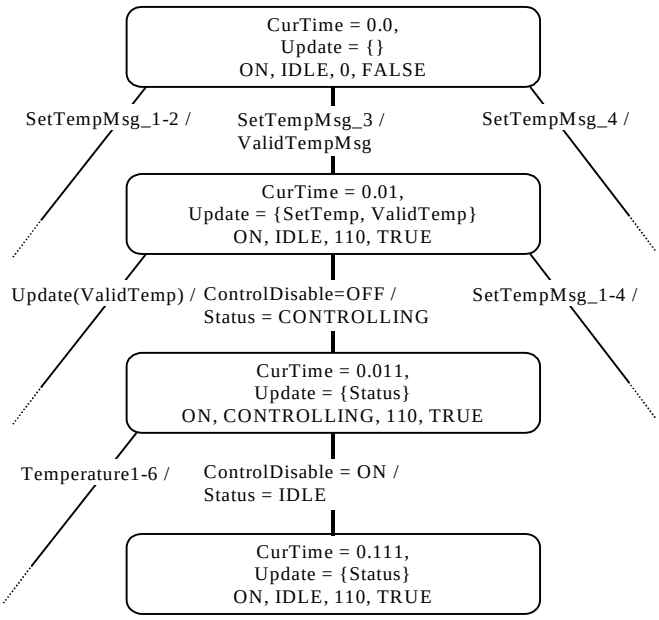

Figure 3. Generated Event Scenario

The update of *ValidTemp* will cause the function *StatusControl* to be evaluated. This evaluation however does not produce any system outputs or further changes to the system state. As a result, the availability fields are checked against current state. Now *SetTempMsg*, and *ControlDisable* are both available inputs. Selecting the path for *ControlDisable* = OFF, the function *StatusControl* is evaluated again. This time, the system responds by setting *Status* equal to CONTROLLING. Since *Status* is a system output this change is listed as both a system response (on the edge between nodes) and in the newly generated state. The new state has the simulation time set to 0.011s since *ControlDisable* must occur at least 1ms after *ValidTemp* and the Updated list is set to include *Status*.

An update to *Status* does not enable any system functions. For the new state, the only available input is *ControlDisable*. *ControlDisable* = ON will cause *StatusControl* to be evaluated. This evaluation changes *Status* back to IDLE which is a state previously reached

thus terminating this branch of the scenario tree. Figure 3 shows the partial scenario tree.

Individual scenarios are extracted from the scenario tree by reading stimuli and response labels from the edges along each path and annotating with any associated timing requirements. All scenarios may be listed by performing a depth first traversal. Additional post processing of the scenario tree may be performed to limit the set of scenarios based on particular test objectives.

6. Conclusions and Future Work

A structured requirements specification for real-time systems based on a set of forms has been presented. The SRS forms build on current requirements engineering practices used in industry. It should be possible to use the commercially available tool DOORS [22] to provide tool support for the creation and maintenance of the SRS forms.

In addition, a method for generating test scenarios from the SRS has been presented. This method is based on a simple space exploration algorithm that uses a finite state machine representation of the proposed system. The inputs to the algorithm are extracted from the SRS forms and the output is a tree structure that represents all possible event scenarios based on the specification and supplied constraints.

Further research is needed in the areas of tool support for the process as depicted in

Figure 2, extensions to provide test case generation for control-oriented systems and for erroneous behavior at the system interface.

7. References

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

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

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

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

[5] Heitmeyer, C., Kirby, J., Labaw, B., "The SCR Method for Formally Specifying, Verifying and Validating Requirements: Tool Support," *Proceedings of the 1997 International Conference on Software Engineering*, pp. 610-11, Boston, May 1997.

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

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

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

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

[10] Lovengreen, H., Ravn, A., Rischel, H., "Design of Embedded Real-Time Ssytems: Developing a Method for Practical Software Engineering," *Proceedings of the IEEE International Conference on Computer Systems and Software Engineering*, pp. 385-90, 1990.

[11] White, S., "Comparative Analysis of Embedded Computer System Requirements Methods," *Proceedings of the First International Conference on Requirements Engineering*, pp. 126-34, 1994.

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

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

[14] Goldsack, S., Finkelstein, A., "Requirements Engineering for Real-Time Systems," *software Engineering Journal*, 6(3), pp. 101-15, May 1991.

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

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

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

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

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

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

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

[22] Quality Software & Systems Ltd., Oxford Science Park, Oxford, UK. DOORS Reference Manual (V3.0), 1996.