TIME BASED REQUIREMENTS AND PARTITIONING OF SYSTEMS WITH AUTOMATIC TEST CASE GENERATION

by

Tony Carl Ewing

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2008

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Tony Carl Ewing entitled "Time Based Requirements and Partitioning of Systems with Automatic Test Case Generation" and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

_____ Date: 24 March 2008
Jerzy Rozenblit

_____ Date: 24 March 2008
Roman Lysecky

_____ Date: 24 March 2008
Susan Lysecky

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

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

_____ Date: 24 March 2008
Dissertation Director: Jerzy Rozenblit

STATEMENT BY AUTHOR

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

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

SIGNED: Tony Carl Ewing

ACKNOWLEDGMENTS

I would like to take this opportunity to thank Karen for handling everything else while I was writing the dissertation. I would also like to thank my committee for taking the time out of very busy schedules to work with me on improving this dissertation. Finally, I would like to thank Steve Cunning for the original automatic test case generation algorithms which I was able to use as a basis for introducing performance and partitioning.

This work was supported by grant number 9554561 from the National Science Foundation.

Dedication

I dedicate this dissertation to my parents who are always there when I need encouragement or assistance.

Table of Contents

TABLE OF CONTENTS — *Continued*

List of Figures

List of Tables

ABSTRACT

Automatic test case generation is a process that starts with text based functional requirements which are converted to a formal system requirements model. Once the formal system requirements model is created the automatic test case generation software creates a set of test scenarios that will verify that the requirements are all met. The automatic test case generation software accomplishes the conversion in a four step process: create base scenarios, identify unverified requirements, enhance scenarios to cover all requirements and allow black box testing, and then combine the scenarios into a single scenario tree. The automatic test case generation system outputs a set of scenarios by walking the final scenario tree. This dissertation expands on automatic test case generation for embedded systems in two major ways. The first is to extend functional automatic test case generation to allow for time based requirements as first class objects. The second is to use the automatic test case generation system to enable system partitioning decisions. The addition of time based requirements to the automatic test case generation system allows more complex systems to be developed. By providing a partitioning recommendation based on the test cases generated from the system requirements, the scope and capabilities of a single designer can be expanded to more complex systems. The resulting upgrades to the theory of automatic test case generation could be applied to the existing tools or incorporated in modern UML/SysML based design tools.

CHAPTER 1

MOTIVATION

1.1 Introduction

Starting in the mid 1990s, the design of computer-based systems has expanded rapidly into new products. As a larger number of engineers are developing computer-based systems, they have the question of how to know that the system is behaving as expected.

Since the complexity of these systems is increasing at an accelerating rate, the number of designers and time required to develop a product is also growing. The pressure of maintaining a performance advantage in commercial (or military) systems leads to the question of how we can reduce the time and cost of developing a system that functions and performs as expected.

Let's evaluate these statements in more detail. What is a *computer-based system*? A system is defined as an object that has expected input and output behaviors. When these behaviors are implemented in part by using at least one digital processing element the system becomes a computer-based system. A computer-based system could be as simple as a toaster with a single digital timing circuit, or as complex as a distributed network of sensors providing border protection for a country. Now what is meant by *function* and *perform*? The generally accepted definition of *function* is that the actual output of the system matches the expected outputs for a given sequence of inputs to the system. When referring to functions, there is generally no concept of time other than the causality of an output occurring in response to an input or some sequence of inputs. The *perform* part of the system behavior defines when the output should occur in relation to the input to the system. For the system to be said to be performing correctly,the system must

have the correct outputs with the proper timing. The behavior of a system is defined as simultaneous consideration of function and performance. Previous work in automated test case generation [1] developed a method for determining that the function of a system matched the expected functions and a basic exploration of the performance of a system. This work will be extended by the proposals in this research.

Next, consider the phrase *as expected*. *As expected* can have many different interpretations depending on who is being asked. There are two main questions that come from defining as expected. The first is, ”Did we build the system right?” Does the system have the behavior that matches the requirements for the system? This question corresponds to verification of a system. It is generally accomplished by testing or analyzing the system as compared to a requirements document with ”shall” statements on a one by one basis. The second question is, ”Did we build the right system?” This question is much more difficult to answer and corresponds to the generally accepted definition of validation. In the case of a commercial product, this question is answered by how many of the product are sold (and used) by consumers. In the case of a military product, this question is generally answered by a formal evaluation process and field use of a system.

Another major concern for both commercial and military system developers is the phrase *reduce the time and cost of developing a system.* The recent acceleration of system complexity has led to an increased emphasis on concurrent engineering and more varied development teams. With the reduced time allowed for development, it is common to have subsystems being designed and developed before the full system requirements are completed. The subsystem designers work with preliminary requirements sometimes with only the functional portion of behavior documented. The performance portion of behavior is estimated by the sub-

system and risk is accepted that some rework may be necessary. The amount of time saved for the entire development cycle by overlapping subsystem development with system requirements can be large, especially when upgrading systems or creating new systems based on previous designs. A good example is estimating the amount of processing throughput and latency required, then including a design margin for the processor requirements. The availability of multiple speed and power grades for general purpose processors allows development of a computer system with the final selection occurring later in the process. The processor boards may be over-designed for the final processor speed selected, but the overlap in time is worth a little extra nonrecurring development cost to allow the board to work at a higher speed than the final requirement. The balance of cost and time are the main issue with the push for concurrent engineering. A partially automated method for generating good partitions of systems early in the design process will help manage the risk of concurrent design of multiple parts of a system before the requirements are finalized.

In a modeling approach to system evaluation and design, a digital model of a real or proposed system is created. For an existing system, the outputs of the digital model can be compared to the real system to verify that the model matches the real system. For a proposed system, the digital model is compared to the current requirements of the proposed system. Previous work [1][2][3] provided a process flow for converting requirements into simulation models with which a proposed system design could be evaluated. In an accelerated system design, the requirements themselves are still in the process of being updated. The behavior of the requirements simulation may lead to changes in the requirements. The first step in this current work is to deeply evaluate the impact and types of performance or time based requirements on the requirements simulation modeling process. The

second step is to propose and evaluate a process by which an early partitioning of system requirements to subsystems can occur based on the existing automatic test case generation, to allow larger teams to work on the same project.

1.2 Context

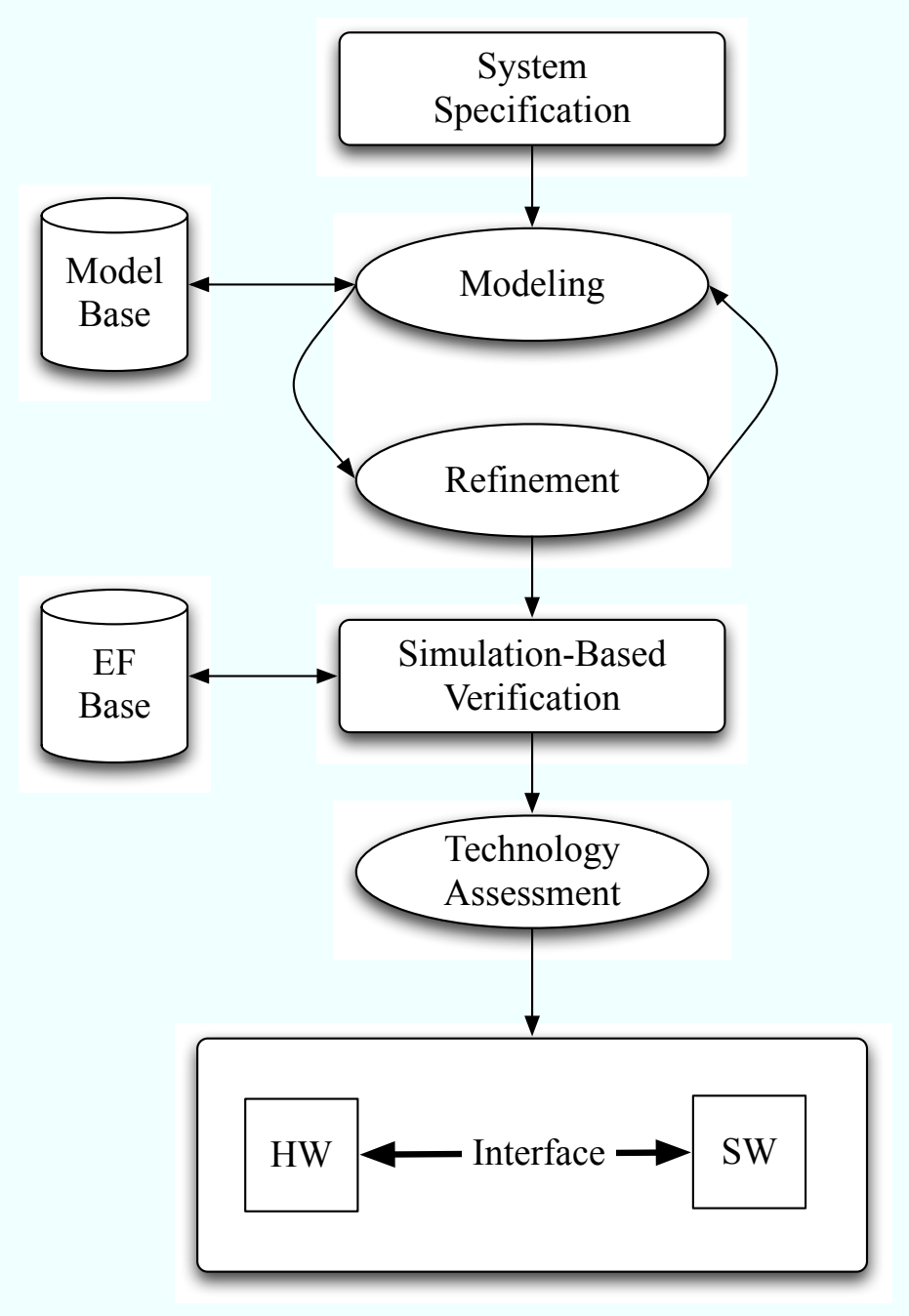
The next sections cover the background materials and definitions necessary to fit this dissertation into the context of current research in the area of automatic test case generation.

1.2.1 Model Based Codesign

Model Based Codesign [3][4] is a method of designing computer-based embedded systems that uses computer simulation and models for converting from an informal system specification into a realizable design model. Starting with an abstract system model, the design is refined in a step-wise manner until enough detail is available to partition the system into hardware and software functionality. Once the partitioning occurs, the testing of models is expanded to compare results with the system implementation. If the model and implementation do not match, the designer evaluates which is exhibiting the expected behavior and updates the other system. An overview of this flow is shown in Figure 1.1 on page 15. A major feature of model-based codesign is that the models are executable and allow for simulation based verification of the system under design. The standard method used in previous codesign works [5] is discrete event system specification.

Discrete Event System Specification (DEVS) is described in [6]. DEVS is an executable specification which is based on time extended, finite state automata. DEVS is a modeling language that is easily used for the creation of system or requirement design models. It has been shown that DEVS is equivalent to the UML

Figure 1.1 Model Based Codesign



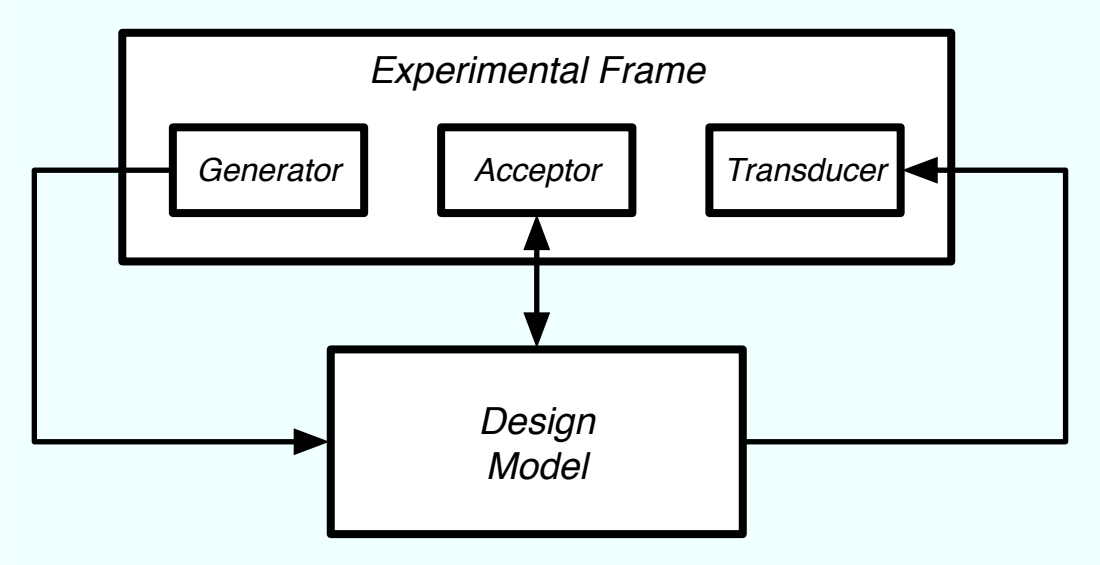


Figure 1.2 Experimental Frame

selected State Charts in representational power [7]. In using DEVS and Model Based Codesign, the System Under Test (SUT) is combined with an Experimental Frame (EF) representing the environment in which the SUT is embedded. An EF consists of three major parts as shown in Figure 1.2. The first part is the generator which creates the inputs to the SUT. The second part is the transducer that records the outputs of the SUT. The final part is an acceptor which determines if the SUT is meeting the requirements tested by the experimental frame. The simplest method for creating an experimental frame is to have a set of scripts that represent test-based behavior that is a simple generator. In previous work [1], a script EF was created using the C/ATLAS IEEE Std 716-1995 [8] language. The next step is to have specific timing in the EF script which allows for more detailed timing to be verified. Another advancement is to include reactive programming in the experimental frame, which introduces communication between the transducer and generator within the experimental frame allowing the experimental

frame to change based on the design model outputs. A well defined model of the environment makes the reactive programming possible. The final step in EF complexity is to include reactive and time based programming in the EF. This makes the EF on the same order of complexity as the system model.

Creating models of systems allows for design to occur with step-wise refinement of the models until an implementation is completed. The models provide a method of verifying that the behavior of the final system matches the expected behavior as specified in the requirements.

1.2.2 Software Cost Reduction (SCR)

SCR was created as a method for creating system requirements for software based systems. Starting in the 1970s, it was realized that the requirements for software based systems were a major cost driver. A study in 1993 determined that a majority of errors in system implementation were actually caused by requirements errors or ambiguities [9]. The SCR* Toolset was created at the Naval Research Labs in a group led by Heitmeyer, and has seen ten years of use solving requirements specification issues for the military. The SCR* Toolset is still being actively improved as described in [10]. In SCR, a system is defined as input (or monitored) variables, processing, and output (or controlled) variables. The relationships among these items are defined by *modes*, *terms*, *conditions*, and *events*. A *mode* corresponds to a state in a state machine. A *term* is a combination of input variables, modes, or terms that allow for a concise specification. An *entity* is defined as an input or output variable, mode, or term. A *condition* is a predicate defined on one or more entities in the specification. An *event* occurs when any entity in the system changes value. An SCR specification consists of a set of tables describing the modes, mode transitions, conditions, and events in the system. The SCR* Toolset allows the specification to be model-checked for incon-

sistencies and completeness. A more complete description of SCR is included in Chapter 2 and [11]. SCR was used to create the requirements model for automatic test case generation algorithm in [1], which is the basis for this dissertation.

1.2.3 Automatic Test Case Generation

The inspiration for this research is described in [1]. The previous work starts with text based requirements and defines a process flow and algorithms for automatically generating test cases based on creating an SCR model of the system requirements. After creating the SCR model of the system, a four part algorithm is executed to automatically create a scenario tree that describes how to test the system under design. The four parts of the algorithm are base scenario generation, expansion to identify unverified requirements, black box testing enhancements, and combination, which generates a final scenario tree. After the scenario tree is generated, a manual process is used to create a C/ATLAS [8] script that can be executed by an experimental frame described in [2]. The parts of the algorithm were formally described as:

1. SGA Scenario Generation Algorithm - Base Scenarios
2. SVA Scenario Verification Algorithm - List of Unverified Requirements
3. SEA Scenario Enhancing Algorithm - Scenario Enhancements
4. SCA Scenario Combining Algorithm - Final Scenario Tree

1.2.4 Time Based Requirements

According to Dasarathy [12], timing requirements have four forms:

1. Stimulus to Stimulus (S→S)
2. Stimulus to Response (S→R)

3. Response to Stimulus (R→S)

4. Response to Response (R→R)

A simple example of each type will help to clarify how these impact testing of a time based requirement based on an automated teller machine. S → S: After the first key is pressed, the last key press of the pass code shall be entered within twenty seconds. S → R: The system shall indicate successful pass code entry within one second of the last key press of the pass code. R → S: After receiving an indication of successful pass code entry, the user shall enter the transaction type within thirty seconds. R → R: Once the system has completed the transaction, it shall return the card to the user within three seconds.

Of these types, R → S and R → R are system requirements while S → R and S → S are requirements on the environment of the system. The current method described in Steve Cunning's dissertation [1] can generate simple S→S type requirements. With additional post-processing it can also determine if the other type of requirements were met. If a R → S requirement on the experimental frame was not met, the test scenario could be modified by hand and run again. This is due to the test scenarios being statically generated. If an R→S requirement is associated with an interval, such as a handshake that must occur within 50 ms, the statically generated test scenarios may not meet the R→S requirement because it cannot know when the response occurs. The smaller the time interval on the requirement, the harder it is to get a static timing to correctly represent the requirement to the system.

1.2.5 Temporal Logic

Since a primary goal of this research is to incorporate performance based requirements into test case generation, an evaluation of methods for representing time

was completed. A recent survey by Bellini [13] was invaluable in learning about temporal logics and their application to specifying real-time systems. The major items of concern were how to represent time, the structure of time, and the metric of time in a specification.

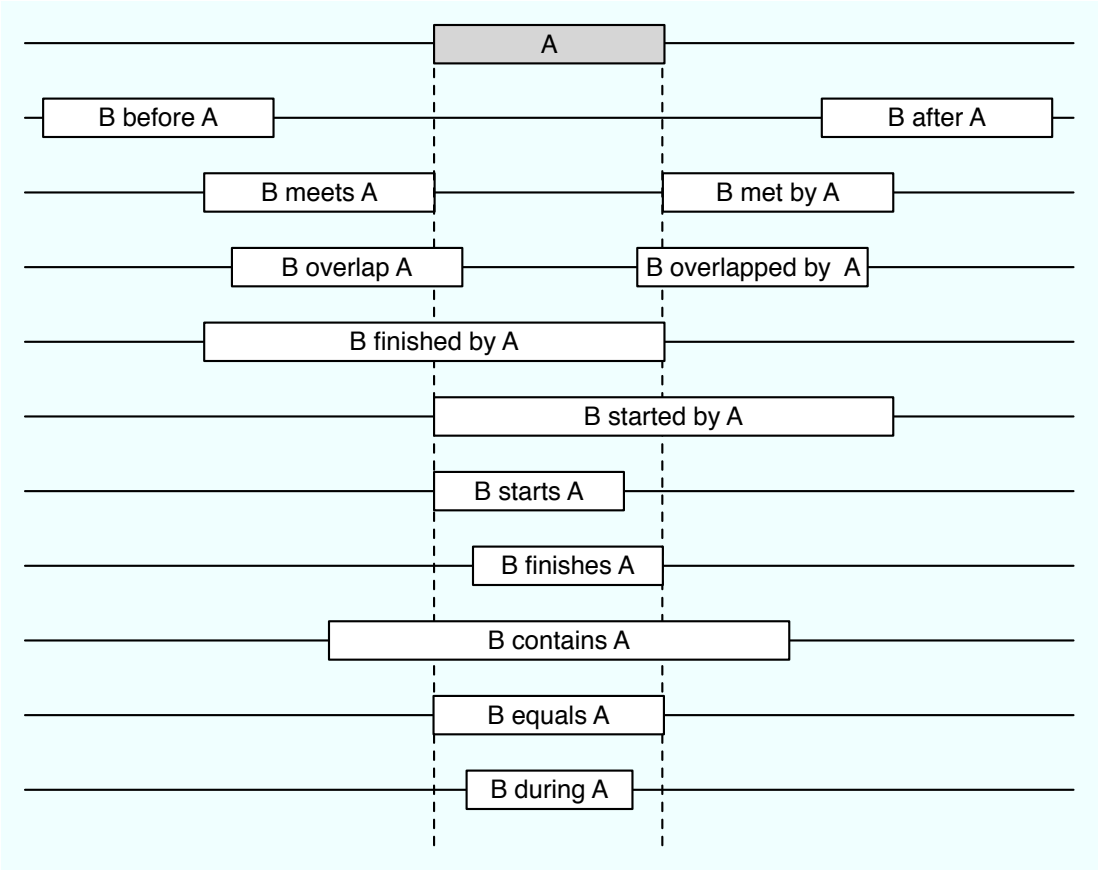
Temporal logics are extensions to classical logical systems that were first developed in the late 1960s and first used in computer science in the 1970s [14] [15] [16]. The use of temporal logics within the formal methods community has a long history that continues to the present time[17] [18] [19] [20] [21] [22] [23] [16] [15] [24]. Temporal logics generally introduce four new operators for reasoning about time. These operators are:

- G - always in the future
- F - eventually in the future
- H - always in the past
- P - eventually in the past

These operators are similar to the $\exists$ and $\forall$ in classical logical systems. In addition to defining these four operators, temporal logics also generally introduce the concepts of *until* and *since*. A *until* B is defined as true when B will become true in the future and until that time, A will always be true. C *since* D is defined as true if D was true in the past and since that time, C has been true. Until and since are useful as they can be used to define the four general operators for a temporal logic.

The fundamental representation of time can be either point or interval based. A point based time system represents time as specific instants that have no length. Intervals are represented by a list of points in time. This is in contrast to interval based logics which represent time defined over an interval. Intervals can be

Figure 1.3 Relationships between Time Intervals



converted to points by defining their length to be zero (or very small). Using intervals also introduces a complication in terms of defining before, after, and at the same time. A re-creation of Figure 3 from [13] in Figure 1.3 shows the many possible relationships between two time intervals. In terms of representing real-time systems, constraints are usually specified in intervals when an event should occur. If a specific time is given, then there is generally a tolerance representing the timing accuracy of the system. Since an interval can be collapsed to represent a point, intervals are a more powerful representation. Intervals are also a natural, logical fit for representing real-time systems.

There are two main ways to represent the structure of time in a temporal logic.

Time can be linear, which corresponds to an intuitive sense of time. There only exists one past and one future extending however far the timeline is defined. In embedded systems, there is generally some reference start time when the system is first turned on. The other method of representing time is to have a branching representation. Branched time means that given the current instant there could be more than one possible future (or past) to get to (or have gotten to) the current state. A branch in the future means that the next state of the system at that branch exists, but is nondeterministic as to which path will be followed. In real-time systems, nondeterminism in the next state is generally considered an error in the system specification. An example of this occurring would be two inputs arriving at exactly the same time, and the system not having a precedence defined, meaning that the implementation of the handling of those two signals could cause one to be lost, or randomly select one of the inputs to handle first. Based on the complexity of branching time logics, at this point it makes sense to use linear time based logic for embedded real-time systems.

The metric for time quality is based on whether or not a quantitative comparison can be made to a time. Temporal logics without a metric for time describe a system as an evolution of events including precedence and cause-effect relationships. A metric for time allows for the creation of a bounded operator such as $F_{[2,4]}A$ which means that A is eventually true in the future between 2 and 4 time units from now. Attempting to specify real-time constraints without a quantitative time is a difficult proposition at best. Since a goal of this research is to allow practicing designers to create better specifications, a system that allows for specifying time quantitatively should be selected.

In summary, a logic that uses intervals with linear time structure and a metric for time is the most useful for specifying embedded real-time systems. These

properties of a temporal logic allow for an intuitive representation of the important properties of embedded real-time systems. An intuitive representation of time is important for achieving acceptance of the proposed methods with practicing engineers.

1.2.6 Time Based Formal Methods

In the past few years, model checking of formal system models has been an area of research in computer science [19] [25] [23] [26]. Clarke reviewed the state of the art in model checking [27] and came to the conclusion that:

> It is possible to use symbolic model checking to verify discrete real-time systems. However, the model checking tools described previously in this book are not suitable to perform this type of verification. It is difficult, for example, to express complex timing properties. It is possible to express the property that "event p will happen in the future," but it is not simple to express the property that "event p will happen within at most n time units" without using the next time operator in convoluted ways. Moreover, quantitative information such as response times or the number of occurrences of events cannot be directly obtained using these techniques. Temporal-logic model checking cannot be used in a natural and efficient way to verify many types of real-time systems that occur frequently in practice.

The SPIN method developed at Bell Labs [28] has been successfully used to test and verify software based systems. The SPIN method is used for verification of concurrent software processes. Since SPIN is specifically designed for verifying software-only systems, it is difficult to use in embedded systems that have a large dependence on hardware interaction and hardware partitioning.

Why should testing be used instead of formal methods? In general, the formal methods determine that a problem exists, but they do not show which part of the code is causing the problem. Table 1.2.6 on page 25 summarizes the advantages and disadvantages of each method as described in previous work in the same area [1]. This dissertation addresses one of the contraindications for testing by evaluating which system internal interfaces should be exposed to an experimental frame level to support hierarchical testing of system designs.

Once the evaluation of time based requirements was complete, the next question is in what other ways automatic test case generation can support the development of larger and more complex systems.

1.3 Trends in Partitioning

Partitioning is the process of dividing a system into more than one component. This process of converting from a single monolithic entity to a group of modules working together to implement a solution is generally a very manual one. The general method for partitioning is to create a graph with vertices and edges, representing system parts and communication paths, then partitioning well-connected groups from each other. A partitioning algorithm uses a cost function that is defined based on which edges are cut and which vertices end up together in the final partitions. Codesign based design methods make the base assumption that part of the system will be implemented using hardware and part will be implemented in software. If any part of the system is implemented in software, then by default there must be at least a selection as to which processor should be used to execute the software. The hardware portions of a system may be further partitioned into smaller blocks to fit into a specific FPGA. A survey paper [20] of embedded systems design methods covered the current methods of partitioning systems. The

Formal Methods		Testing	
Pros	Cons	Pros	Cons
Correctness guaranteed	Requires knowledge and creation of formal specification	Applied at multiple levels of design, can check consistency between specification levels.	Correctness not guaranteed (Cannot simulate all possible errors)
Can be applied early	Can be computationally complex	Can be applied early	Can be computationally complex to generate
May be applied hierarchically	Inconsistencies are flagged, but not isolated to specific cause	Builds confidence in early designs	Can be expensive to apply
	Formal specification is software based, assumes that hardware is correct	Tests may be reused for regression testing	Testing may be difficult due to interface accessibility at system level
		Tests can be applied to modules	Cost of test maintenance

Table 1.1 Formal Methods vs. Testing

methods in use were implemented using both hardware and software modeling languages, including data flow graphs, hardware description languages, and finite state machines. This large variation is due to the many different methods of system design that exist and are being investigated currently. The partitioning methods were also implemented at many different levels of design from early in the process with hierarchy to late in the process with operation based partitioning [29]. The algorithms used in the partitioning process were general graph based algorithms or by hand. The graph based algorithms included simulated annealing [30], min-cut [31], clustering [32], and Kernighan and Lin [33]. An important feature of a partitioning algorithm is the cost function that determines how good each possible partitioning is defined to be. The cost functions included code profiling, scheduling analysis, communications cost, concurrency allowed, execution time (SW) and area (HW). This wide variety of cost functions implies that partitioning methods are still in the early research phases for codesign based systems. A close examination of the partitioning methods explored reveals that most of them are implemented during the system synthesis phase, not during the early design and development. Part of the software design process is to partition the code into logical units that can be compiled (and more importantly, tested) independently. The partitioning aspect of the software design process is currently so embedded in the system design that it is not seen as a partitioning method to be explored. What would happen if, instead of partitioning a system based on the design specification, we partitioned a system based on the requirements specification? Many of the current partitioning methods include communication costs as part of their cost function. The communication cost is a side effect of the system architecture and the specific implementation of the system. With an appropriate requirements partitioning between components, the communication

cost may be driven to zero. If it is not zero, it can then be traced to a specific system requirement or partitioning decision if the partitioning is done at the requirements level instead of the implementation level. In practice, partitioning is accomplished by designers' performing functional decomposition and assigning different functions to different components in the system based on their previous experience. These decompositions may not be optimal or even very good for a new set of requirements, but they allow for creating similar systems with a lower risk.

1.4 Problem Definition and Goals

There are two main goals for this work. The first is to extend functional automatic test case generation to allow for time based requirements as first class objects. The second is to use the automatic test case generation system to enable system partitioning decisions.

Automatic test case generation includes a method for including very simple time-out based timing requirements. However, more complicated systems generally require complex timing behaviors to be correct. The previous test case generation allowed for testing that the right actions occurred, but not that they occurred in the right time frame.

As systems grow larger and more complex, it is standard practice to partition the system into subsystems that are then responsible for combining together to meet the whole system requirements. Currently industry practice [34] [35] [36] for partitioning large systems relies on experienced systems engineers and functional allocation of requirements to subsystems. A major issue with partitioning is cleanly dividing the requirements into independent subsystems for verification. A rule of thumb based on experience with the testing of complex systems is

that final integration and verification may be up to one third of the entire schedule. As systems are partitioned into subsystems, there arise derived requirements based on the interface created between two (or more) subsystems to meet overall system requirements. The goal for this partitioning algorithm is to cleanly assign requirements to subsystems, and recommend subsystem integration orders based on the automatic test case generation graphs.

CHAPTER 2

TIME BASED REQUIREMENTS

2.1 Time in Automatic Test Generation

This chapter describes an extension of the automated test generation system as described by Cunning [1] to include complex temporal requirements. Specifically, it proposes how to incorporate timing requirements into SCR tables used to describe the system being designed. A full description of the SCR method of specifying systems is described by Heitmeyer in [9].

Two ways to handle time in SCR specifications were introduced by Cunning in [1]. The first is to add an explicit time reference such as T_{ref} which is incorporated into the system interface, and the second is to include a monitored variable *Time* which is an artificial input to the system. Both of these methods require careful handling to make sure the introduction of time does not corrupt the system under test.

Adding a variable such as T_{ref} to represent time requires that the development process enforce the rule that access to T_{ref} is only allowed if the system has implemented an internal global clock. If the system is partitioned or distributed, the use of a global clock may corrupt the implementation and make it unrealizable. This issue is well known in the model based design community and the process for verifying models includes checks to make sure that only information available through sensors or communications paths are made available to the system models. The following section demonstrates how to use time as an implicit part of the system design rather using an external input.

References to time in the text based requirements are identified by the requirements writer who is converting the design into the SCR notation. Very early in

the design, it is useful to create an SCR representation of the system requirements that only include the functional requirements. Once the functional specification has been checked using the SCR tools, the set of requirements is known to be consistent. By then generating a set of scenarios using the SGA, the system designers have a set of simple test cases for their early designs. The early designs are generally at a very high level and either do not have time included, or simulate time. Examples of a high level early design would be state machines, Unified Modeling Language (UML) diagrams, or flow charts. Since the scenarios created by the SGA guarantee full coverage of the input requirements (or a list of unverifiable requirements), the early designs can be functionally verified while the performance requirements are being developed.

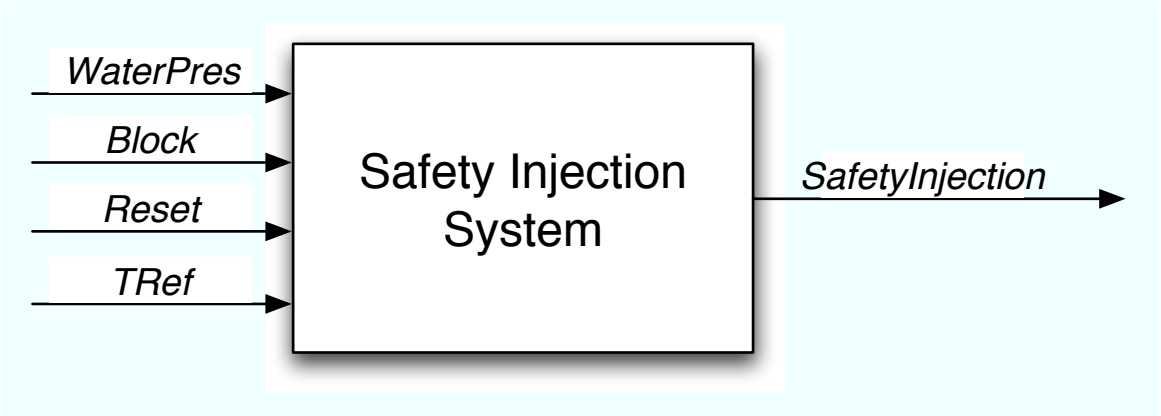
As a system design matures, it is more important to verify that the performance requirements are being met as soon as an implementation decision is made. The earlier that a performance requirement is tested, the cheaper it is to make a change if the requirement is not met by the current implementation. The scenarios generated by a modified SGA would include enough information to create an acceptor in the experimental frame. It is important that the performance based scenario trees are available before the system design is converted to a low level design. The process of converting the design to a more detailed level involves selecting implementation processors, infrastructure, and languages. If the system designers only have the functional specification, they could choose to implement a time critical function in Java only to find out that the performance based requirements of completing the function within 30 μs wold have been more suited to a hardware based solution. While creating the functional requirements first is ideal to allow some early design, it is critical that the creation of the performance based requirements occur before the system design becomes too detailed.

In addition to having the performance based requirements, it is important to have an experimental frame that is capable of measuring performance based requirements. In an ideal situation, the experimental frame would include an online acceptor for all the system requirements. These potential effects on the experimental frame will be discussed after the evaluation of how time is to be included in an SCR model. The performance requirements are created by looking for certain key words in the requirements text including within, after X time units, or before X time units elapse. Since this process of conversion is being completed manually, a tool could highlight suspect phrases which the designer then adds to the SCR model.

2.2 Safety Injection Example

Since this work is expanding on the work of Cunning described in [1], it will use the same example problem of the safety injection system for a nuclear reactor which was based on the example in [37]. The goal of the safety injection system is to ensure that the water system cooling the core of a nuclear reactor maintains a safe operating temperature and pressure.

Figure 2.1 Safety Injection System Block Diagram



The text based requirements for the system are shown in Table 2.1 on page 33. An implementer's view of this system is a group of state variables Pressure, Overridden, SafetyInjection, and TrefCnt. Pressure is the variable that represents WaterPres and has two possible values of TOOLOW and PERMITTED. The demarcation line between the two states is based on the threshold defined by the value LOW in the description. Overridden is defined as a boolean variable whose value is dependent on the Block (True) and Reset (False) signals being asserted by the operator. If Overridden is true, the SafetyInjection output is disabled even if the value of Pressure indicates that SafetyInjection should be On. SafetyInjection is an output of the system with the values On, indicating that water should be added to the cooling system, and Off, which indicates that the cooling system water level is acceptable. When SafetyInjection is On, water is being added, which increases the WaterPres. TRefCnt is an integer variable which counts the number of times TRef has occurred since the last assertion of Block by the operator. Note that [R4] has used TRef as an extra input which is not really part of the system. It also adds a table to the system for handling TRefCnt and TRef. The initial state of the system is defined to be {Pressure, Overridden, TRefCnt, SafetyInjection} = {Permitted, False, 0, Off}.

Now, we will walk through the SCR specification for this system, starting with the mode class table for Pressure shown in Table 2.2. The first column represents the current mode for the system variable. The second column is an event which triggers the mode to change, and the third column is the new mode for the system variable. The final column is a text field which is used in the automatic test case generation system to identify requirements in the scenario trees. So the first row says, "When the Pressure mode is TooLow and the WaterPres becomes $\geq$ to LOW, the system should change Pressure mode to Permitted." This mode tran-

Table 2.1 Text Based Requirements for Safety Injection System

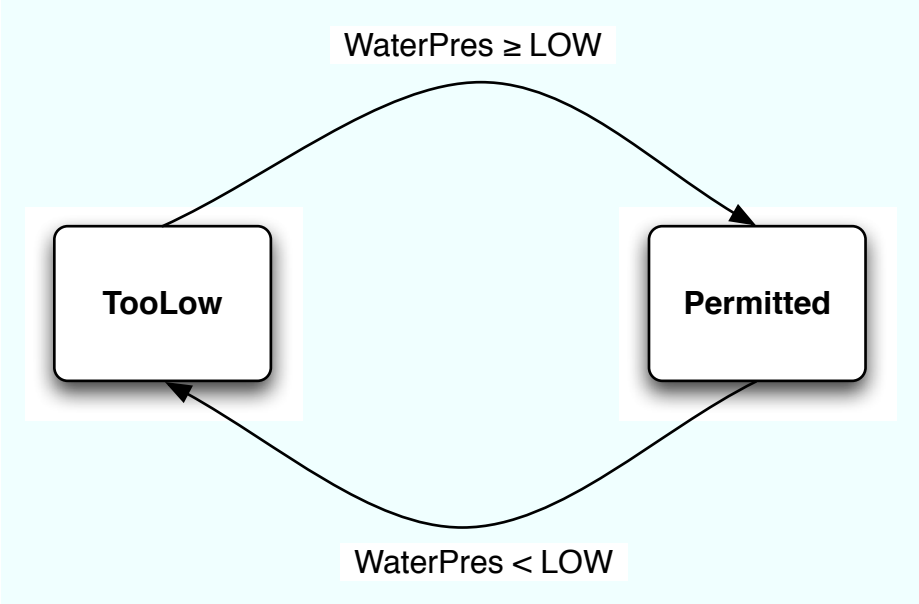
$[R1]$	The system shall assert *SafetyInjection* when *WaterPres* falls below LOW.
$[R2]$	The system shall be considered blocked in response to *Block* being asserted while *Reset* is not asserted and *WaterPres* is below LOW, and shall remain blocked until either *Reset* is asserted or *WaterPres* crosses 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*.

sition table is equivalent to the state diagram shown in Figure 2.2 on page 34. As a reference on notation, @T(A) means that event A has become true, @F(B) means that event B has become false, @C(D) is the time at which D changes, and E′ is a new value for a variable in the next state for the system. The other special notations are @T(False) which is any entry to indicate that that change never occurs (False never becomes True) and @T(Inmode) which occurs when the system enters that state described by the Mode at the left of the table. The formal representation for SCR is described in [11].

Table 2.2 Mode Transition Table for Pressure

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

Figure 2.2 State Diagram of Pressure



The next portion of the SCR specification is an event table for the variable

Overridden which is shown in Table 2.3. The goal of the event table is to describe how a variable changes based on the state and input events to the system. For this example, if the system mode is TooLow and Block occurs while the Reset input is Off (@C(Block) When (Reset=Off)), the new value of Overridden is True. Similarly, if TRef occurs when the internal variable TrefCnt is 2 (@C(TRef) When TrefCnt=2), the new value of Overridden is False. The event table for TRefCnt shown in Table 2.4 works the same way, expect instead of true or false, it has a simple formula the the new value of TRefCnt is the old value plus one.

Table 2.3 Event Table for Overridden

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

Table 2.4 Event Table for TRefCnt

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

The final part of the SCR specification is a condition table which defines the

output of the system based on the system mode and internal variables. In this example, if the mode is permitted then the new value of SafetyInjection is Off. On the other hand if the mode is TooLow and the Overridden is false, then the new value of SafetyInjection is On.

Table 2.5 Condition Table for Safety Injection

Mode	**Events**			
Permitted	True	[R3a]	False	
TooLow	Overridden	[R3b]	NOT Overridden	[R1]
$SafetyInjection'$	Off		On	

2.3 Time in SCR

As discussed previously in Section 1.2.2 on page 17, SCR specifications consist of modes, terms, conditions, and events. Each item requires inspection to determine what type of timing requirements should be specified.

Modes define the state of a system and as such do not require any specific timing. A mode class table is used to define the mode transitions, so it is logical to include mode based timing in the mode class table. An example of a mode based requirement would be if the system is in *waiting* for more than three seconds without any events occurring, transition to *idle*. This would allow the system to change into a low power or sleep mode if there are not relevant events occurring. In cases where the system is allowed to go to *idle* there would probably be a companion requirement such as, if the system is in *idle* it must respond to *Important Event(s)* within 40 milliseconds and all other events within 1 second. A requirement like this would require a test case for every event in the system, after

the system has gone to the *idle* state. The mode transition introduces the keyword *within* for defining the amount of time before the transition must complete.

Terms are combinations of input variables, modes, or other terms used to simplify a specification. Since terms are where the entities are combined together, it is the logical place to include a timing component. The current formal specification for terms is boolean expressions on conditions and events. This can be expanded with timing qualifiers from temporal logics. These qualifiers include *for* timing interval and *within*. In combination with the *for* and *within* keywords, the logical combinations using and (&) and or (|) allow for representing nearly all timing based requirements.

Conditions are predicates defined on entities in the system. Since conditions include terms, which include timing notation, conditions will indicate when time has elapsed, but they do not need a new method of specifying timing. The impact of conditions occurs within the experimental frame under which a system is being tested. The complexity of the conditions as a whole defines the amount of time required for the experimental frame to execute. In the current system with a static C/ATLAS file, the complexity of the conditions leads to more time being required to evaluate the results of executing a test script. Example C/ATLAS test scripts are included in Appendix B, starting on page 89.

Events occur when any system entity changes value. Since conditions can be used to detect an event occurring, a new condition can be created based on a timeout from an event. This means that there is no modification necessary to the specification of an event for it to be used to define a time based requirement.

Based on the analysis of the pieces of an SCR specification, only mode transitions and terms need to have a method for specifying time. Pulling all the changes together: The syntax of the keyword *for* is A *for* Interval. The meaning of this

Table 2.6 Mathematical Definition of Intervals

$[t_0, t_1]$	$t_0 \leq T(E) \leq t_1$
(t_0, t_1)	$t_0 < T(E) < t_1$
$[t_0, t_1)$	$t_0 \leq T(E) < t_1$
$(t_0, t_1]$	$t_0 < T(E) \leq t_1$

statement is that A is true during the entirety of the specified time interval. The syntax of the keyword *within* is B *within* Interval. The meaning of this statement is that B is true at some point during the specified time interval. To be complete according to temporal logic [13], the two operators *since* and *until* must be defined. The derivation of *since* is X time since A becomes A *within* $[t_0, t_0]$ and !A *for* $(t_0, t_0 + X]$. The derivation of *until* is C *until* D becomes C *for* $[t_0, t_1)$ and D *within* $[t_1, t_1]$.

2.4 Time Specification

As discussed in Section 1.2.5, the natural representation of time in real-time embedded systems is using intervals. How can a time interval be represented? The simplest method is to use the mathematical interval notation to represent time. The mathematical notation is simply a set of two numbers with square brackets or parenthesis indicating a closed or open interval respectively. The interval definition also specifies that the first number is less than or equal to the second number. The example of all four possible combinations and their meanings with T(E) being the time of the event of interest are shown in Table 2.4.

If the time units are not specified, then they would default to the system default, or to seconds if there is not a system default specified. The lower limit would be zero and the upper limit could be infinity. Examples where $T(E)$ is the

time of the event occurring are : $(0, 1)$ meaning $0 < T(E) < 1$, $[5s, 7s]$ meaning 5 seconds $\leq T(E) \leq$ 7 seconds, or (0,1] meaning $0 < T(E) \leq 1$. If there is no time requirement on a specific transition, then it has a timing spec of [0,1) which would mean that the transition must occur before the next time step for the system which makes sense for a discrete time system. For discrete event systems, the default timing spec would be $[0, t_a)$, meaning the transition must occur before the next time of interest to the system. For continuous time systems, there is generally a highest frequency of system response, so [0,$\frac{1}{f_{max}}$) would be a logical choice for the default timing specification.

A simple model of stimulus or response being a single event is very easy to implement by adding a timing specification to each table entry in the SCR description of the system. A timing entry in a Mode table indicates how long the system has to execute the state transition. A timing entry in a condition table indicates how much time is necessary to resolve the timing specification in that condition transition.

Each entry in an SCR table has a unique identifier attached by the method described in [1], which allows for the creation of more complex timing requirements. Any combination of base requirements connected using & or | can be used for a more complex requirement. For example, [R1] and [R2] shall occur within 50 ms of [R3] occurring. This example is really specifying a timing-only requirement that refers back to behavioral requirements. This type of requirement does not cleanly fit into the SCR tables, so it will be added as a comment to the top level system description. Using the same notation as the condition tables, this would become "$[R3] => [R1]\&[R2] within [0, 50ms]$." The system timing requirements can include references to requirements, events, conditions, or terms defined in the SCR tables.

The use of intervals allows for sequential events to be defined as shown here: A *within* [t_0,t_1) & B *within* [t_1,t_2]. Using t_1 as the endpoint for A and the starting point for B means that A must occur before B. If desired, the symbolic value for t_1 can be left in the specification to indicate that as long as A occurs first, it doesn't matter where in the interval [t_0,t_2] A occurs as long as it is before B.

Based on the analysis in the previous sections, there is now a way to include performance based requirements in the SCR version of the requirements specification. A mode table will have an added comment with a time interval to indicate how much time is allowed before the mode transition is completed. Terms are now allowed to include references to the new keywords *for* and *within*, which allow for describing relative timing between events. The use of *for* and *within* would require changes to the SCR* Toolset, so an alternative method would be to have only the default SCR behavior in the table, with a comment including the full performance specification. Other performance-based requirements are added as comments in the system description section. These changes allow for the continued use of the base SCR* Toolset to perform automated checks for a variety of design errors including:

1. syntax and completeness - all tables have syntactically correct entry in all locations

2. disjointness - nondeterminism is not allowed

3. coverage - state dependent assignments are defined for all appropriate states

4. state reachability - there are not unreachable states

5. cycle detection - prevent an infinite loop of state changes

2.5 Modifying Text Requirements to Include Timing

The modified system requirements using timing would appear as shown in Table 2.7 with the simplified block diagram in Figure 2.3. When modifying the requirements to use specific timing, it was noted that there is an ambiguity in the text due to using *Block* as an input and blocked as a system state. The ambiguity lies in what happens if the *Block* input occurs while the system is blocked. It could mean that 3 time units from entering blocked the system unblocks, or 3 time units from the last *Block* input occurred the system unblocks. To resolve this ambiguity, the event table for TRefCnt (Table 2.4) was consulted. In the event table it is clear that TRefCnt is set to 0 when the *Block* input occurs. The modification to the text requirement is to change [R4] to refer to "three seconds since the last *Block* input". This modified version of the text removes the artificial TRef variable, and clears an ambiguity in the text based requirement.

Figure 2.3 Simplified Safety Injection System Block Diagram

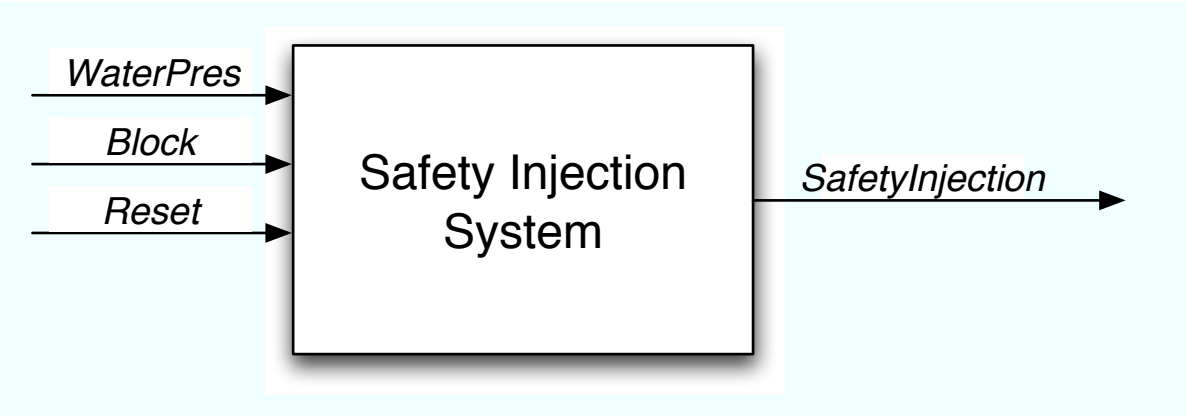


Table 2.7 Modified Text Based Requirements for Safety Injection System

$[R1]$	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*~~ three seconds since the last *Block* input.

2.6 Timed Requirements in SCR

This section describes how to modify the original SCR specification as shown in Section 2.2 to include the time based requirements. The state of the system was defined by:

- Pressure $\in$ {Permitted, TooLow}
- Overridden $\in$ {True, False}
- TRefCnt $\in \{0..3\}$
- SafetyInjection $\in$ {On, Off}

The initial state of the system is defined by a 4-tuple representing the values of each variable,{Pressure, Overridden, TRefCnt, SafetyInjection}, in this case defined as {Permitted, False, 0, Off}. Note that the variable Overridden is equivalent to the term Blocked in the text based requirements. Now, what is the impact of adding an implicit timing model to the SCR notation? As noted in Table 2.7, all references to $TRef$ as an input to the system, and $TRefCnt$ as an internal variable have been removed from the text based requirements. The first major change is that the $TRefCnt$ Event Table 2.4 is no longer necessary to describe the system. Since this system is based on discrete time semantics, the logical choice for a default transition is [0,1), meaning that the transition must occur before the next time step for the system.

Table 2.8 Modified Mode Transition Table for Pressure

Old Mode	**Event**	**New Mode**	
TooLow	@T(WaterPres $\geq$ LOW)	Permitted	[P1]**[0,1)**
Permitted	@T(WaterPres $<$ LOW)	TooLow	[P2]**[0,1)**

Table 2.9 Modified Event Table for Overridden

Mode	**Events**			
TooLow	@C(Block) when (Reset = Off)	[R2a]**[0,1)**	@T(Inmode)	[R2b]**[0,1)**
TooLow	@T(False)		@T(Reset=On)	[R2c]**[0,1)**
TooLow	@T(False)		@T(Overridden) & 3 since @C(Block)	[R4~~d~~] **[3,4)**
$Overridden'$	True		False	

Table 2.10 Modified Condition Table for Safety Injection

Mode	**Events**			
Permitted	True	[R3a]**[0,1)**	False	
TooLow	Overridden	[R3b]**[0,1)**	NOT Overridden	[R1]**[0,1)**
$SafetyInjection'$	Off		On	

For the example of reactor safety, almost all the requirements would have the default discrete timing spec of [0,1) and the timeout for [R4d] would become [3,4). The table for handling TRef and TRefCnt is removed from the system and the event table for Overridden is changed to Table 2.9. The meaning is now that if the system is in state TooLow and Overridden is true with 3 (seconds) since the last Block input then before 4 time units in the future Overridden becomes False. Note that [R4] is entirely handled by a single entry in this mode table which is why [R4d] becomes [R4]. If the ambiguity in the requirement were handled in the opposite way, meaning three time units since Overridden was asserted, the entry would be simply be @T (Overridden) with the same timeout value of three seconds.

After the modification to include time as an implicit variable, the system state is defined by:

- Pressure $\in$ {Permitted, TooLow}
- Overridden $\in$ {True, False}
- SafetyInjection $\in$ {On, Off}

2.7 Timed Requirements in Scenario Generation

The automatic test case generation system described in Section 1.2.3 creates essentially two scenario trees. The first tree occurs after the initial SGA with a greedy search followed by a distance based search. Because the nodes that were multiple time reference steps away are now adjacent with a label, the greedy search is now more efficient at finding performance based requirements. Figure 2.4 shows the original results and Figure 2.5 shows how the modified version finds an extra requirement before the distance based search is executed. The differences are

noted by shading the nodes grey. The greedy search algorithm can now generate a simpler sequence without having to generate TRef multiple times using the distance based search. Now each edge in the scenario tree is labeled with a timing requirement that allows the greedy search to be more effective at covering requirements. The simple example described here shows how the number of uncovered requirements from the first portion of SGA will be reduced, making the directed graph search have a smaller set of requirements to cover. This method does not change the extensions to the test generation algorithm to allow for black box testing of the requirements. In many cases where time delays are involved, the scenario enhancement algorithm (for black box testing) will also have a smaller set of requirements to cover. On systems with a large number of timing requirements, this will help reduce the time required to execute the scenario tree generation, since the first algorithm will find a requirement coverage before the other algorithms are executed. The greedy search algorithm will now generate a tree containing all except [R2b] as shown in Figure 2.5. Each edge in the graph is also labeled with a timing interval which will make the C/ATLAS [8] program, or any other experimental frame driver, easier to generate.

In a manual fashion, the rest of the pieces of the automatic test case generation system was executed to show how the final outputs would be modified. The original scenario tree is in Figure 2.6 with the modified output shown in Figure 2.7. The nodes highlighted in grey show the areas of the scenario tree that are different. The main difference is that the three nodes that had *TRef* events were replaced by edges with non-default timing specifications. This matches the expected result of removing the *TRef* and *TRefCnt* in the specification in favor of explicit timing information. Now the question arises of how to handle the timing specification on each edge of the scenario tree.

Table 2.11 State Description for Scenario Tree

State	Pressure	Overridden	TRefCnt	Safety Injection
1 (2,8)	Permitted	False	0	Off
6 (7,9,11)	TooLow	False	0	On
3 (10)	TooLow	True	0	Off
4 (12)	TooLow	True	1	Off
5 (13)	TooLow	True	2	Off

Table 2.12 Modified State Description for Scenario Tree

State	Pressure	Overridden	Safety Injection
1 (2,8)	Permitted	False	Off
6 (7,9,11)	TooLow	False	On
3 (10,13)	TooLow	True	Off

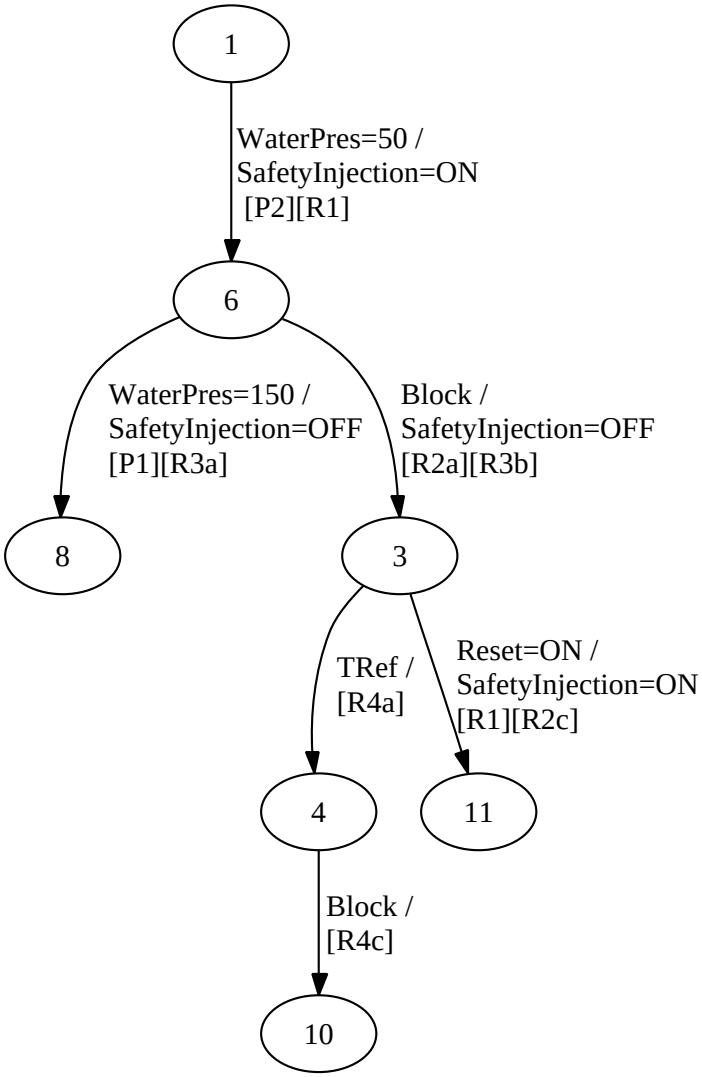


Figure 2.4 Scenario Tree from Original Greedy Search

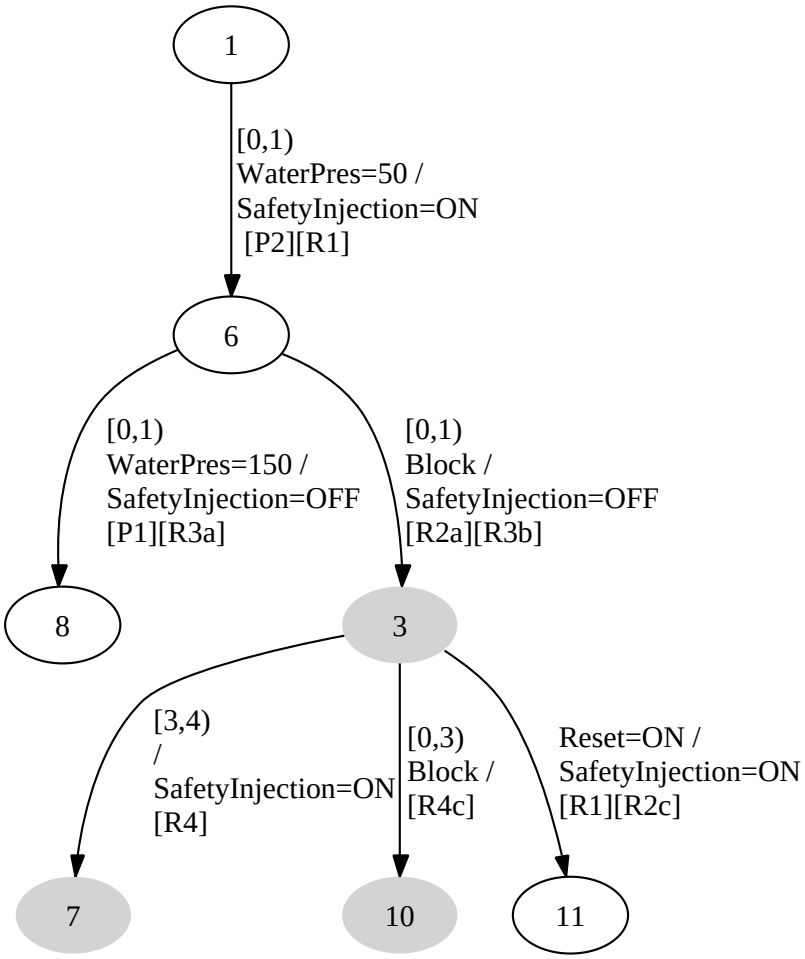


Figure 2.5 Scenario Tree from Modified Greedy Search

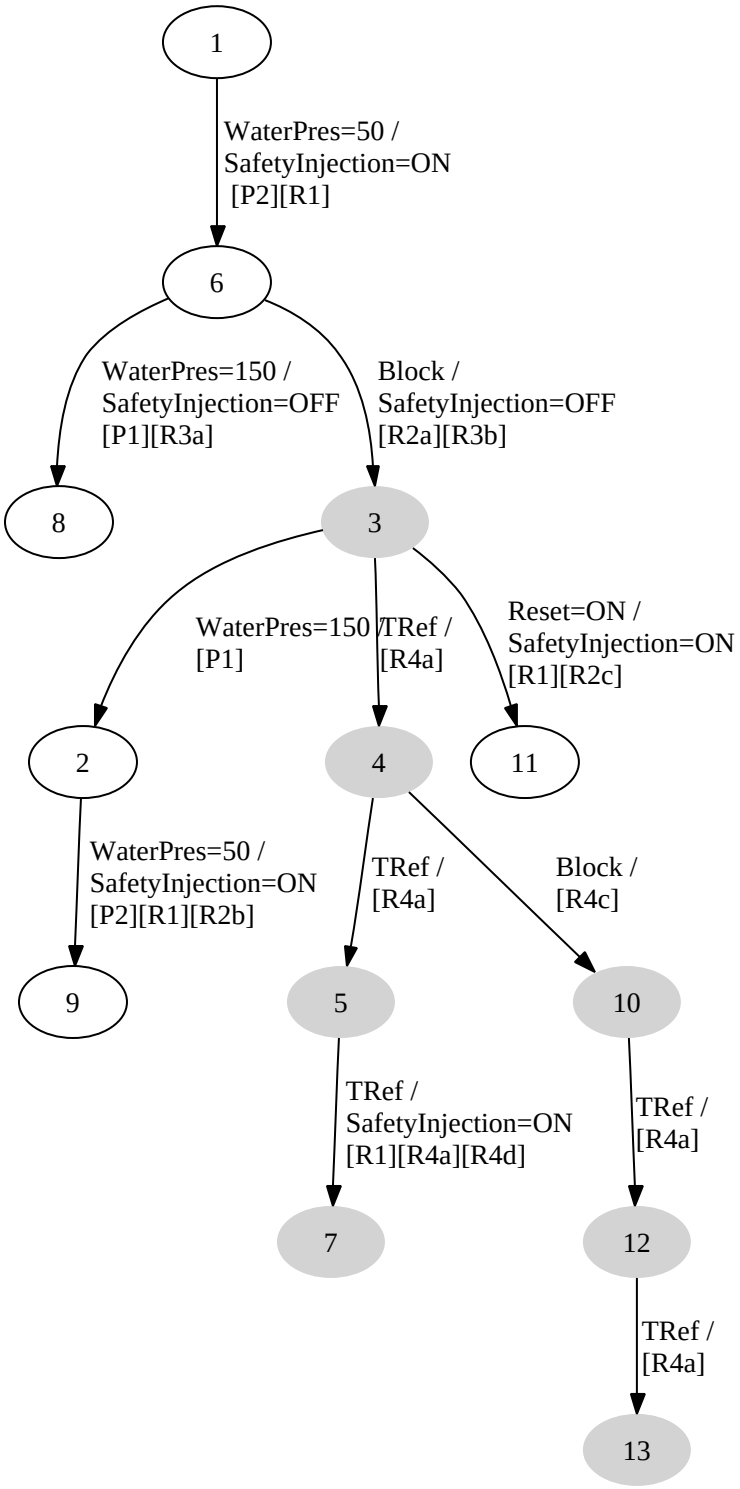


Figure 2.6 Scenario Tree from SGA

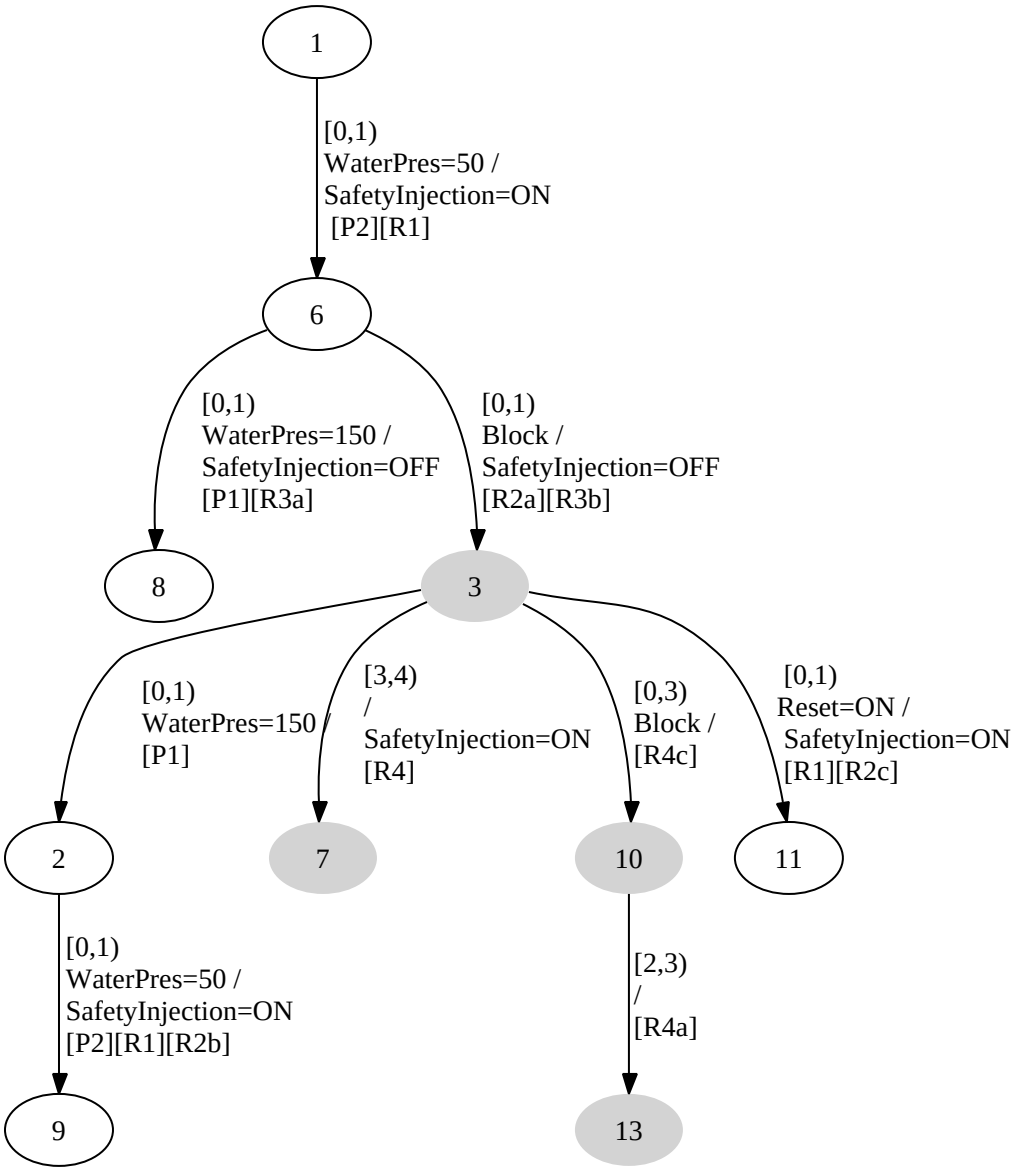


Figure 2.7 Scenario Tree from SGA with Time Updates

2.8 Conversion of Timed Scenario Tree to Experimental Frame

During the early design phases, where the specification is still functional in nature, it is best to simply use the original automatic test case generation system. For examples of C/ATLAS[8] scenario files from the original automatic test case generation system, see Appendix B starting on page 89. Since this process was manual, the next step would be to explore how it could be automated.

Previously, the possible labels on an edge included INEVENT/OUTEVENT where either the INEVENT or OUTEVENT could be empty. This line was converted to two lines in the C/ATLAS file as shown here:

```
APPLY, INEVENT
WAIT FOR, EVENT OUTEVENT, MAX-TIME 1 TIME UNIT
```

If INEVENT is empty, the first line is removed. If OUTEVENT is empty the second line is changed to :

```
WAIT FOR, 1 TIME UNIT
```

Based on these rules, it would be possible to automate the creation of this portion of a C/ATLAS scenario file.

Now, we proceed to the question of how to handle a modified scenario tree with timing specifications on each edge. If there is a default timing specification on an edge, then the timing can be handled in the exact same way as above. If no output is specified, then a

```
WAIT FOR, 1 TIME UNIT
```

command is inserted into the test scenario. If an output is expected, then a

```
WAIT FOR, EVENT OUTEVENT, MAX 1 TIME UNIT
```

command is inserted into the test scenario. If an edge has a timing specification such as [A,B] then a the new command to insert is:

```
WAIT FOR, A TIME UNITS
WAIT FOR, EVENT EXPECTED EVENT, MAX (B-A) TIME UNITS
```

Putting it all together, an edge labeled with a timing specification of

```
INEVENT/OUTEVENT[A,B]
```

is represented by

```
APPLY, INEVENT
WAIT FOR, A TIME UNITS
WAIT FOR, EVENT OUTEVENT, MAX (B-A) TIME UNITS
```

in the test scenario file. This covers the addition of simple time-based requirements to a functional requirements specification.

There is also the addition of complex time based requirements including the X *for* [A,B]. A complex requirement may have edges that are labeled with embedded timing specifications that are difficult to handle due to overlap. If the specification includes the possibility of overlapping events as described in Section 1.2.5, these requirements would generally not exist in early versions of the specifications, but would appear as the design matures into an implementation. If the system has the property of partial time ordering it is likely that a full system level test may require interleaved or overlapped timing on some of the inputs. Each subsystem could be tested independently, but when the overall system is combined together, the interleaving would exceed the capabilities of a static time based test script. Also remember from Section 1.2.4 that the C/ATLAS test scripts are only designed to generate the Stimulus to the system under test and depend

on the Experimental Frame C/ATLAS execution engine to also record outputs for Stimulus to Response (S→R) and Response to Response (R→R). Since the previous Experimental Frame engine [2] was designed to execute from C/ATLAS scripts, it does not have the capability of generating Response to Stimulus (R→S) inputs to the system under test without tuning to a specific design model. The limitations of the previous system lead to an investigation of other possible solutions for creating the experimental frame based on the Scenario Generation Algorithm outputs.

2.9 DEVS Model for Timing Properties

With additional post processing, the algorithm can also determine if the other types of requirements were met. If a R→S requirement on the experimental frame was not met, the test scenario could be hand modified and run again. This is due to the test scenarios being statically generated and not observing the system output timing to generate the timing for the next input. If an R→S requirement is associated with an interval, such as a handshake that must occur within 50 ms, the statically generated test scenarios may not meet the R→S requirement because it cannot know when the Response occurs. The smaller the time interval on the requirement, the harder it is to get a static timing to correctly present an input to the system. The test script may also require modification each time the system implementation is updated, which may slightly change the timing behavior of the system under test. This section demonstrates a modification on the method created for the Automatic Test Case Generation system described in [1], [2], and [38] to create a reactive Experimental Frame in response to timing requirements. Based on experience with developing experimental frame systems, the new system is a reactive DEVS model that includes the capability to generate

R → S requirements and online determination of R → R and S → R timing. By having online detection the system is able to change from behaving similar to a compiler to behaving like a debugger, which makes analysis of system failures easier. A compiler only detects static errors related to the syntax and semantics of the system specification such as incorrect static type assignments and unused variables. A debugger allows the user to create breakpoints and examine the state of a system that is executing. As a result, this allows for its internal behavior to be examined during the early phases of system development. As the design is implemented in real hardware, it becomes more difficult to stop the system and inspect the internal state, unless this is specifically built into the system. An experimental frame that is used in a debugging style will help to point out internal variables that are useful for evaluating system behavior and should be exposed to the test environment in the system implementation.

In the original automatic test case generation system, the output of the scenario generation algorithm was manually converted to a C/ATLAS script file. Instead of using the manually generated C/ATLAS script, it is possible to directly use the Scenario Generation Algorithm output file with a DEVS experimental frame system. This is the final output after the algorithm has accomplished black box style testing of all the system requirements. The output file consists of 5 major sections as shown in Appendix A.4 on page 85. The first section is the title and status section, which indicates if any part of the algorithm was not successful. The next section is a table of the system states. The header on this section indicates each variable name, with the table delineating the value of each variable for each of the defined states. Note that the states are numbered for uniqueness, and equivalent states that were found in the tree expansion are marked. These equivalent states will be useful later in converting back to a graph for creating a

partitioning of the system. After the state table come the scenarios, which represent requirements traces through the system. The fourth section of the SGA output is a list of the edges in the scenario search tree. These edges are uniquely numbered and are specified by the use of an ordered set of previously defined states. The final section of the SGA output is a statistical summary to allow for tree characteristics and run time to be monitored for SGA performance. The main piece of information missing from this file is a symbol table for the variables defined by the system. The expected values for each variable are listed in the state table, but the range of possible values is not listed. It is also possible to determine from the graph edge labels which variables are inputs or outputs from the system.

This issue complicates automatic generation of a DEVS experimental frame from the test output file. A good first estimate of variable types would be to identify integers and floating point numbers, then assume all other values are enumerated types. Each scenario listed in the file would become a DEVS model to run against the system design model. Appendix A.4 is an example output from the automatic test case generation algorithm.

The first step in this process is to create a simple DEVS model for the experimental frame that reads in the current specifications, which are currently in the C/ATLAS language. Example of the C/ATLAS scenarios are included in Appendix B. The following section walks through the process of creating a DEVS model from the C/ATLAS test script.

Each line has a unique number for reference in the output of a test script. Numbers in the 1000 range are the header information with the name of the program and the types of signals being used. Numbers in the 2000 range are used to define the signaling interface. These translate to ports and data types in a DEVS

model. The items in the 3000 range are output events. These allow a C/ATLAS script to wait for an output before moving on to the next input. These are all events that the reactive DEVS experimental frame must identify to handle R $\rightarrow$ S events correctly. A C/ATLAS test program simply lists a maximum wait time that it will wait before checking if the event occurred. Next, items in the 4000 range get the system into the initial conditions and the 5000 range items set up the event monitoring. All this information is repeated for each scenario that is generated by the scenario generation algorithm. It is easy to see how a DEVS model can use this information to set up the experimental frame system. Now the only information that is needed is expected behavior of the system under test. This information is contained in the 6000 range elements for each test. The first DEVS experimental frame would include an initializer that would be called with the scenario number to drive the model and record the input and output sequence. This is a replication of the work previously done, simply using DEVS for the experimental frame instead of the C/ATLAS test driver, thus allowing for the testing of more complex requirements in the second phase of implementation.

One disadvantage of the current system is that a C/ATLAS WAIT FOR command has limitations in the type of requirements that it can test. It must wait for a specific event, so if there are multiple possible responses it does not work. Handling an *or* type requirement correctly requires more intelligence. Another limitation is waiting for two events to occur, but not knowing the order in which they might occur. These two simple improvements to the types of events can also be combined making complex wait statements that cannot be handled correctly by C/ATLAS programs. A DEVS model with a simple online transducer can wait for a complex condition to be met. This demonstrates the benefits of using a DEVS model as the experimental frame. Any type of wait can be ac-

complished by using code inside the DEVS model. The code can come directly from the requirements model and be added to the experimental frame. By using a DEVS model for the experimental frame, it also makes the application of R $\rightarrow$ S requirements easy to include with dynamic timing. This complex experimental frame starts with the input-output specification based on the C/ATLAS test generation, but would add the complex timing requirements in code from the original requirements model done in SCR.

In the previous work [2], the system design model was implemented using StateMate^{TM}by iLogix. In related work [7], it was shown that DEVS and Statecharts have equivalent representational power. By using DEVS for both the scenario and the design model, it would simplify troubleshooting and debugging of the final system. DEVS has also been expanded to include some real-time testing functionality [39] since the previous work was completed. The DEVS prototype for the system design model could be easily created at the same time as the experimental frame. The creation of a prototype for implementing the design model ensures that the experimental frame and design model will agree on the environment interface, which means that executing the tests is simplified. Based on the system interface as defined in the requirements, a DEVS model would be created with the appropriate ports defined for system inputs and outputs. This DEVS model would be the base to start the system implementation models. As the system model gets converted to real hardware, the test driver code uses interface code in the DEVS system model to drive the real system interfaces. The interface code would be kept in a modeling library that allows for the reuse of existing interface code with new system designs. As more systems are designed using this system, the task of creating the interface code would be simplified to a selecting appropriate interface models from the library.

Ideally, the final version of the DEVS experimental frame would include the full requirements model and the tree of requirements generated by the scenario generation algorithm. It could then automatically run all the tests necessary to verify if a system has the correct behavior and performance. Since SCR is a table based method for representing state machines, it can be represented using the DEVS modeling formalism. This would allow the full SCR system requirements model to be embedded in a DEVS acceptor model. The DEVS acceptor could perform online evaluation of the design model outputs and immediately flag inconsistent outputs for debugging while the system state is still available. In order to perform all the system tests, the system allows the design model to be reset to its original state. This involves either deallocating a system model and then instantiating a new model, or control of the system power later in the design process. Another method of accomplishing this is to include a checkpointing system at the high level DEVS model for the system. Checkpointing is not currently evaluated, but checkpointing techniques are now widely used in grid and cluster computing environments.

The output of the experimental frame would include both the expected behavior and the system behavior, with an annotation showing discrepancies. Starting with a system model, the experimental frame could stop time if an online check of expected versus actual behavior exists. This feature of online debugging to catch errors would require that time could be stopped, which is easy to accomplish in a simulation of the system. As the design progresses towards a final implementation it would require a special time stop function that could stop the hardware time counters, similar to a JTAG/COP port on some current microprocessors. In general, this capability would not be needed as the difficult timing issues would be known and worked out before beginning the implementation of the system in

hardware (or software). The timing issues that exist in the simulation will more likely lead to design for testability, which will allow access to the internal signals necessary to verify that the performance is being met in a physical implementation. Since these issues are known well before the hardware is being designed, it will be easy to include access to the required signals without expensive rework of hardware.

This chapter has covered the process of incorporating performance based requirements into the automatic test case generation system. It started with a treatment of incorporating time into the text based requirements, updating the formal SCR model for the system, and demonstrating changes to the Scenario Generation Algorithm suite. After updating the test cases to include timing, it described improvements to the C/ATLAS driven testing described by Cunning and Jaroch in [38]. The first minor improvement is to run C/ATLAS test scripts from within a DEVS experimental frame which reduces the overhead of running multiple test scripts to running a single model which automatically generates all the script output. The second improvement is to handle the more complex R $\rightarrow$ S and combination type events by using a DEVS experimental frame. The final improvement is to include a full SCR requirements model in the DEVS experimental frame so that it can perform online checks and annotate the output log or allow for live debugging of the system being designed.

CHAPTER 3

TEST CASE BASED PARTITIONING

This chapter proposes a new method of partitioning early in the design process using the generated scenario test tree. The current methods of partitioning as described in [20] all operate on the design specification. Requirements based partitioning is done in practice through the creation of a system architecture that defines major functions with communication patterns. These methods rely on the experience of the system designer with designing similar systems to create a good functional decomposition of the design and then assigning requirements to different components based on what has worked well in the past. In approaching the process from a requirements perspective before implementation is started the goal is to find a partitioning of requirements that allow for easier testing and verification of the final system design. If test cases can be separated by allocating an entire requirement to a single component in the system, then none of the other components need to be tested to verify that requirement. System safety based requirements are key candidates for partitioning into components, while security based requirements are poor candidates. If a safety requirement is partitioned, then the design of that portion of the system can use more formal version control and process requirements to guarantee safety requirements are met. At the other extreme, security based requirements are usually emergent properties and the implementation depends on the whole system being aware of the security requirements. Both of these requirement types need to be very carefully tested at the system level with, at a minimum, the full automatically generated test cases.

3.1 Partitioning Implications and Examples

In the ideal case, partitioning would take the set of requirements and group them into disjoint sets with no communication required. If an ideal partitioning occurs, it is due to the system actually having multiple components that are not interacting, and they could be independently designed and built. More likely, though, there are missing or misunderstood requirements in the system description. In more realistic cases, some requirements will need to be partially allocated to two (or more) subsystems. These will generally require the addition of derived interface requirements so that B and C can meet the total requirements specified in A. $A => B + C$ in the ideal case becomes $A => B + (B$ interface with $C) + C$ realistically. Examples of this include pipelining stage buffers in processors, serial communications between sensors and control computers, and TCP/IP layering.

3.1.1 Pipelining

Pipelining is an example of performance based partitioning. The pieces of a task are broken into discrete steps such as Fetch and Decode which can be designed and developed independently, but operate in parallel. Pipelining trades between latency and throughput. The latency of any specific instruction is generally larger with buffer waits between stages, but more instructions can be being performed at any one time. Pipelining is generally a performance based decision, but can be used to partition different types of computation when throughput is a concern. How can potential sites for pipelining be identified from the system requirements? Pipelining is a good candidate when the individual traces on the test requirements are very long sequences with little branching. These long sequences of steps should be investigated to see if they are truly sequential with dependencies between steps, or if they can operate in parallel. If they are se-

quential, then pipelining is a good method for increasing performance. If the operations are parallel, then adding independent subsystems may be a better choice. Pipelining introduces a handshake or synchronization so that all stages complete and move results to the next stage without losing intermediate results. The buffers and communication between stages can be exposed to the experimental frame for the system as a whole. Outputs from a previous stage can then be simulated, which allows each stage of the pipeline to be tested independently. Pipeline stages should have requirements described with transformations of inputs to outputs for each stage being well defined. If a system transformation is broken into multiple stages, then the overall requirement should be allocated to multiple stages, with the derived transformation at each stage showing the expected inputs and outputs for that stage with a pointer back to the original requirement.

3.1.2 Specialization and Distribution

Having specialized sensors and control computers is an example of specialization and distributed processing partitioning. By having a well defined serial interface to the primary computer, the system design allows for easy replacement or upgrading of sensors. Each sensor implementer has the responsibility of understanding in detail the limitations and strengths of their chosen sensing device. The serial interface allows a central control computer to receive input at known intervals and spend its time implementing overall control algorithms for the system. By having a simple serial interface, the sensor subsystem can have a very simple processor with a single task that operates at all times. In many cases, this style of specialization is assumed before starting the central computer design. The safety injection system, from the earlier example, assumes there is a sensor input for water pressure that has been evaluated before the requirements are ex-

ecuted. The evaluation of the sensor input could occur on the same processor as the rest of the safety injection requirements, but if it did, the requirements for the sensor would need to be included in the safety injection requirements, as they will have an impact on the timing of the safety injection system. Once a scenario tree is generated, specialization can be recognized by repetitions in the tree. Another classic example is the control system for an elevator. The doors open and close on each requested floor. The process of opening and closing the door can be partitioned into a component with a simple interface of ”open door request” and ”door closed” indication to the main algorithm. The scenario tree for the floor decision portion is simplified by not showing how the door is opened and closed with wait states each time it selects a floor. This simplification speeds up even an implementation of the floor decision algorithm, because it can be driven by a test harness that speeds up the door open/close process. Within the open and close door process it is now easier to evaluate the safety criteria of not closing a door on a person by allocating the safety requirement to a smaller subset of the whole system. The safety aspect of closing the doors is now handled in a subsystem and would not need to be checked on every floor in the elevator design. The safety aspect can be fully tested by checking in the full system on only one floor because the implementation of the safety feature is independent of the floor at which the elevator has stopped.

Another type of partitioning is subtree partitioning. If a requirement or group of requirements only appears in one branch of the generated scenario tree, then it can be handled by creating a separate subsystem that only handles the requirements in that subtree. This type of partitioning is a special case of specialization where the requirements are only covered in one branch of the generated scenario tree. This is easier to recognize than the pattern subgraph matching required for

full specialization partitioning as it only requires starting at the leaf nodes and counting the number of times each requirement appears on each edge on the way up to the root of the tree. The implementation of subtree partitioning leads to a system design with communicating independent components at a high level in the design. The requirements that appear only in a given subtree are handled by a single component, while the other requirements that appear in multiple subtrees are handled by the communication or control coordinator in the system. Since this method is easier to recognize than full specialization, it is a good candidate for initial automation.

3.1.3 Layering

The final type of partitioning identified by examining the scenario tree is to layer functionality so that the software is responsible for calculations and writing a value to a register, while the hardware is responsible for using the register value and creating a voltage output. In this case it is very likely that one requirement will be distributed to both a software and hardware component. In this type of partitioning of a requirement, it is best to declare the two components as part of a larger subsystem that is independently verified. The software cannot be completed until the interface to the hardware is defined. The system design can affect the experimental frame by having a layered design style, which is very common in large distributed systems. Each distributed subsystem will have a stack of components that implement a communication network at the bottom. A layered design style leads to layers that can be implemented and tested independent of the other layers being finished, as long as the interface between layers is well defined. An example of this design style is a set of sensors in a network. The network layer is designed to pass messages from node to node, and this can be tested without the presence of the final packets that a sensor will send. For exam-

ple, sensors can be tested on a standard UDP/IP network before their implementation network is completed, this will not have the same final performance, but it can verify the functionality of the design. By marking requirements with the layers they belong to, or as a layer to layer interface, it would allow earlier testing of the layers. Each layer could be manually marked, and then a separate scenario tree could be created to test each layer. This has the advantage of identifying layers that are not completely specified before a large amount of design work is expended. This type of partitioning is not easy to identify from the scenario tree. In general, computer-based systems always have at least two layers, one that represents the hardware and one that represents software. Software designs usually implement a layered architecture, with an operating system providing services to applications. The operating system layer is not specifically described in the requirements, but as system behavior becomes more complex, testing that multiple processes execute and run, meeting all deadlines, requires the support infrastructure of an operating system.

3.1.4 Requirement Model Implications

The requirements created by partitioning need to be marked as derived, so they are allowed to be changed if a partitioning decision changes. Derived requirements can be seen as negotiating points for the components involved in implementing the interface. These derived requirements allow each subsystem to be tested independently with an augmented system experimental frame before final integration occurs. This augmented experimental frame allows for simpler fault insertion testing and exhaustive message testing that may not be possible with the final system components. Since the interface between subsystems is exposed to the experimental frame, it is possible for the experimental frame to identify which side of the interface is violating its requirements. If neither side is violat-

ing its requirements, then the derived requirements must contain a specification error. It is also possible to eliminate some test scenarios, as described in the elevator example above. A requirement was fully allocated to a subsystem that was independent of the floor of the elevator, so the requirement only needs to be tested on one floor to be verified for all floors. When comparing two different partitions of the same design problem, the original function and performance scenario tree is used to evaluate the system designs against each other. This is done so that the augmented requirements are not converted into system requirements, which limit the designers' freedom in partitioning the original problem. The augmented requirements, including derived interfaces, are useful for debugging a specific design, or for creating subsystem replacements when improving a current design.

Based on the expansion of the C/ATLAS program to a DEVS experimental frame described in the previous chapter, how can an experimental frame be created for partitioned systems? The first method to consider is to make no changes to the EF generation algorithm. The time based EF will be used at some point to evaluate an implementation of the system which does not allow for anything except the defined environmental interface to the system. Another method would be to require any implementation to expose the defined subsystem interfaces based on requirements partitioning to the EF. This method appears to be a large imposition on the system implementation, but it only specifies that derived interface requirements are not hidden from a test environment. If the system is truly implemented with the derived interface, there will be either software variables or hardware lines to implement the interface. If all the pieces of a system end up implemented as a System on a Chip (SoC), then the derived interface requirements would point to the exact items needed to verify the correct operation of each sec-

tion of a system implementation. Forcing the system implementation to expose derived interface requirements, which first appeared to be a large imposition, actually leads to a testable design that has clearly defined roles for each component in the design. The final evaluation of a system with respect to its functional and performance requirements should use the original system scenario tree to ensure that the system meets the original requirements, and to evaluate the system performance. If the evaluation indicates that it meets the performance requirements with the extra debugging interfaces intact, it is ideal to leave these items in place in the final system unless security is the primary concern. By leaving the debugging interfaces intact, information can be gathered to allow system states to be recreated in a controlled environment with the implementers present. Since the full test cases are accessible, it can be determined from the debugging information which test case is failing in the field, and determine the root cause of the failure in an expedient manner.

3.2 Algorithm

In the previous partitioning methods reviewed in the survey by Edwards [20], the graph created used the implementation of a design to perform the partitioning. In this chapter, the use of the scenario tree to implement a graph based partitioning algorithm on the system requirements is explored. As described previously, the output of the SGA includes a graph whose vertices represent systems states, and whose edges represent transitions between states. The edges of the graph are also labeled with the requirement(s) that are verified by the transition along that edge. The goal of this partitioning is to have a group of requirements handled by exactly one component in the system. Since there are 4 different types of partitions that can be identified from the scenario tree, each type of partitioning will

use a slightly different method to identify the partitions.

The difficult part in creating a partitioning algorithm is creating a cost function that works well. Since this algorithm is partitioning based on requirements coverage, the first part of the cost should be that all edges with a specific requirement are in the same component. For example, every time the elevator door opens and closes in the test cases it should be mapped to the same partition. Another goal of partitioning is to have approximately equal sized components when completed. At one extreme is having each requirement handled by a separate component, which would imply that the requirements are describing a set of loosely coupled components that do not require interaction. A requirement specification with this property may imply either groupings of small systems (a network of identical sensors), or a disjoint set of requirements. At the other extreme is the output comprising a single component, which implies tightly coupled requirements that form a cohesive whole. If the system falls into the second category, it makes sense to start a more detailed design and apply the techniques of synthesis partitioning to break the system into hardware and software components. Each partition type is identified by a specific cost for the partitioning method. These results are then presented to the system designer to choose which requirements are allocated to new subcomponents.

This partitioning may create more than two components, but they should all be approximately the same size. In this case, the size is defined as the number of requirements that are handled by each component. The second part of the cost function is a measure of the side effects of a partitioning. The primary side effect is that unless a perfect partition occurs where there are no additional ”interface” requirements the cost is the complexity of the interface requirements. Since the automatic creation of the interface is difficult, an estimate will be made to assist

in the partitioning. This estimate consists of the number of system state variables that change in separate components. If a state variable only changes in a single component, then it is likely that no other component needs to have the information related to that variable. If more than one component needs access to a state variable, the easiest way to handle that is to define one component as the owner (with the logical choice being the component that changes the variable the most), and the other components then need the changes in the variable to be communicated, making it part of an internal interface. If many components need access to a state variable, it makes sense to encapsulate that state variable into its own component. This encapsulation allows for handling possibly concurrent reads and writes to a state variable in a single component. This component then becomes an active object that is responsible for ordering write requests based on the rest of the design. The active object could still be implemented as software using locking mechanisms, or hardware using a register with atomic access. This early decision that a variable needs to have access controls allows for thorough testing of the access controls before the whole system is integrated. The basic cost is defined by the number of extra requirements to implement the partitioning, along with the standard deviation of the number of requirements per component.

As the system design is implemented, updates can be made to the partitioning costs. The implementation of a specific component may be recognized as similar to an already existing system; since the requirements on the component are clearly defined, it is possible to determine if reuse is feasible. There are three possible conditions: the implementation could be a perfect match, it could be missing one or more requirements, or it could be more than the requirements. For an implementation that is a perfect match, which can be tested using the scenarios defined for the component, the cost for using that partition should be reduced.

If the implementation is missing one or more requirements (which would be the most common case), the components cost should be reduced but still include the cost of modifying an existing system. A human judgement call would be needed to estimate the cost of modifying the existing system, but could be estimated by the number of new requirements the object must satisfy as a percentage of the total number of requirements the object implements. In the final case where an implementation covers more than the requirements, the ideal situation would be to compare the extra covered requirements with the scenario tree to verify if the requirements overlap, or if the implementation provides unnecessary features. If an implementation provides unnecessary features, it would make sense to add the extra requirements to the EF for that component, to verify that it does not break the previous system, but then mark those requirements as reuse. Reuse requirements would have the lowest priority and will probably only be tested at the component level. Remember that in all these cases of reusing previous designs, the Scenario Generation Algorithm has created test cases that can be used to verify that the reused component(s) meets the new requirements. The basic cost is updated with the increase in size of the scenario tree to test all components.

This process of cost updating implies that a library of previous implementations with the requirements covered and test cases would help the requirements partitioning process. The many attempts at reuse have shown that without knowing exactly what requirements were covered by a component, it is very difficult to implement reuse successfully. By implementing and using the Scenario Generation Algorithm then keeping track of the results of implementation, it will make reuse easier. The requirements trace for testing a component can be directly compared with the new requirements traces to determine if there is overlap. Even if the selected requirements trace does not match, a comparison of the requirements

covered directly may show a candidate for reuse. Since there is some randomness in which paths are selected in the scenario tree, the new system may have generated a different path through a group of very similar requirements. The next step is to identify the specifics for each of the partitioning types recognizable from the scenario tree.

3.2.1 Layering

Layering is not automatically identified from the scenario tree since it is based on designer decisions. This design architecture is supported by creating an augmented scenario tree to show the impacts of layering on the system test complexity. The cost function for layering is the number of requirements created for layer to layer interfacing. Layering in designs also allows for the creation of common components that can be used across multiple products. The application layer of software can rely on the implementation of hardware level drivers that provide the same interface for multiple products. Layering also allows for specialization of engineering design talent to hardware specific and software specific functions. Automatic test case generation provides the designer with a cost for the layer selection that is being considered for comparison with other layering possibilities.

3.2.2 Pipelining

Pipelining is identified by an in-order tree walk that annotates the number of nodes that occur without branching. Starting at the root of the tree (which starts with a zero) if the number of children is greater than one, each child is marked with a zero. If the number of children is one, the current marking is incremented by one. After the walk is complete to the leaf nodes, the tree traversal is reversed to the root, and nodes marked with a non-zero number get the marking of their child and are kept track of for presenting to the designer. After the tree walk is

complete there is a list of test node chains to present to the designer as possible pipelines. The chains should be presented with the longest chains first, since those are the best opportunities for pipelining. The designer is then allowed to select how many pipelining stages to include and mark in the requirements. The number of stages selected is directly proportional to the number of communication paths, which represent the requirements cost of implementing pipelining.

3.2.3 Specialization and Distribution

The recognition of specialization and distribution is identical to the process of subgraph matching. Since it is NP Complete [40], any of the heuristics for subgraph matching can be used to identify possible subgraphs from the original tree. In general, the designer will be heavily involved in identifying a subgraph which the search algorithm can then locate in the rest of the scenario tree. This should be implemented as an interactive process where the designer can select a subgraph then be told how many times it appears in the rest of the scenario tree. An automated method would start with all subgraphs of a minimum size, probably five nodes, and then execute the search for matching subgraphs on the rest of the scenario tree. The reason to start with five nodes is to balance the effort to create the partition with the number of requirements contained within the specialized component. The final value for the minimum subgraph size would be determined based on the total size of the system and experimentation with multiple examples. This algorithm is ideal for implementing on a large number of independent computers using the map reduce [41] algorithm. Each computer is given a subgraph and the full scenario tree, and responds with subgraph matches to the reduce computer, which gathers the matches and report them to the designer. The reduce portion could be limited to reporting when at least four copies of the subgraph are in the original tree. The cost function for specialization is the

number of requirements created to communicate with the component minus the number of requirements implemented by the component. This cost encourages larger components with low communication overhead.

3.2.4 Subtree

Subtree identification is separated from specialization, since it can be solved with a polynomial time algorithm. Starting with the leaf nodes, one annotates which requirements are covered by each edge as the tree is traversed to the root node. For each requirement, if only one child has a nonzero value, the parent records a one for that requirement. If all children have a zero value the parent records a zero. If more than one child has a nonzero value the parent records the sum of the child values. If the edge leading to a node covers the requirement, the node adds one to the child's value. Once the root node is reached, each requirement that has a value of one in the root node is covered by exactly one subtree. The subtree with the requirements is identified by the the largest value for that requirement and all its children. The number of requirements with a one defines the number of subtrees. The cost function for subtrees is one over the number of requirements covered by that subtree plus the number of additional interface requirements created.

3.3 Partitioning Results

The use of the scenario tree allows for the identification of partitioning that is not obvious based on the requirements text. The early identification of pipelining, specialization, and subtree partition opportunities allows the system designer to distribute the requirements to components which can be independently designed and tested. Each type of partitioning is independent and can be implemented one at a time. The inclusion of layered partitioning into the scenario tree generation

allows for test cases to be automatically generated for layered design testing. The inclusion of metrics indicating partition quality based on the test cases generated allows designers to make partitioning decisions based on the test implementation costs. The combination of automatic test generation with an experimental frame reduces the total testing cost, which allows for a more thorough test coverage of the original requirements for the same cost. Since the test cases are generated automatically, an estimate of testing time can also be used by the designer for determining the quality of a specific system requirements partitioning before the components are implemented and the cost of component design is expended.

Chapter 4

Conclusions, and Future Directions

The expansion of the automatic test case generation system to add performance based requirements allows embedded system designers to test and verify more complex systems. The changes to the original system allow for the expansion of a scenario tree to include all the system requirements in a test plan before the detailed system design is started. The requirements on a real-time experimental frame are known early in the process, which allows the experimental frame to be ready for testing as the system is designed. This work outlines the method for converting the previously manual experimental frame execution process into an automated process that can be run with updated requirements at any time. Now that the test case generation system includes the functional and performance based requirements, it should be evaluated for inclusion with UML and the evolving SysML. Recent commercial developments in test case generation from UML models can be evaluated in reference to the test scenario generation algorithm for completeness of coverage, test case generation time, and test execution times. The automatic test case generation algorithm could be modified to work with the UML models in addition to the SCR formal method. The major issue for working with UML models and automatic test case generations is that the semantic execution the the UML is not consistent across the entire UML space. The first task would be to evaluate the Model Driven Design subsets of the UML for use with automatic test case generation.

By further expanding the test case generation algorithm to support partitioning, this work has revealed an area for using test case generation on systems that are larger than previously feasible. By allowing the system to be partitioned, the

initial decisions for partitioning a large system can be evaluated and design alternatives explored based on the estimated costs of testing the final system design. By supporting partitioning and augmentation of the scenario tree, this work allows designers to recommend subsystem designs with verification scenarios for inclusion into the overall system. Since the scenario trees can be generated from any subset of the requirements, the subsystem designers know exactly how the rest of the system is expected to behave and they can test before all the components are complete. The proposed partitioning algorithms need to be automated with the scenario generation algorithm in the future. This work in early partitioning needs to be explored with a large number of embedded system designs for a final evaluation as to how effective these specific metrics and cost functions apply to all embedded system designs.

The next step in this research is to update the test case generation algorithm to using the DEVS based requirements model along with the experimental frame automatic generation. Once the DEVS versions of the system exists, a study of a variety of embedded systems should be evaluated using the automatic test case generation system. The implementation of the partitioning heuristics can be accomplished in parallel once the DEVS interfaces for the system requirements models are defined.

APPENDIX A

AUTOMATIC TEST CASE GENERATION OUTPUT LISTING

A.1 Scenario Generation Algorithm

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

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

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

7 UNIQUE SYSTEM STATES HAVE BEEN GENERATED:

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

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

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

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

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

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

```
    10 (3)

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

ST CONTAINS 11 nodes.

SCENARIO SEARCH TREE (16 total nodes):

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

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

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

Total execution time: 0.010000 seconds
```

A.2 Scenario Verification Algorithm

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

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



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

Attempting to verify requirement P2...VERIFIED
Attempting to verify requirement R1...VERIFIED
```

```
Attempting to verify requirement P1...VERIFIED
Attempting to verify requirement R3a...VERIFIED

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

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

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

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

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

```
     3 TRef / [R4a]
     4 Block / [R4c]
    10

Attempting to verify requirement R4c...NOT VERIFIED

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

Attempting to verify requirement R2c...VERIFIED


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

Total execution time: 0.000000 seconds
```

A.3 Scenario Enhancement Algorithm

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

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


PROCESSING SCENARIO      4
```

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

3 UNIQUE SYSTEM STATES HAVE BEEN GENERATED:

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

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

ST CONTAINS 3 nodes.

SCENARIO SEARCH TREE (7 total nodes):

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

```
     4 (  3,  6)  'WaterPres = 150 /'
     5 (  3,  7)  'TRef /'
     6 (  1,  4)  'Reset = ON / SafetyInjection = ON'

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


Total execution time: 0.000000 seconds
```

A.4 Scenario Combining Algorithm

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

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

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

7 UNIQUE SYSTEM STATES HAVE BEEN GENERATED:

 Name Pressure SafetyInjection TRefCnt Overridden
-----------------------------------------------
     1  PERMITTED             OFF           0       FALSE
    8 = 1
     2  PERMITTED             OFF           0        TRUE
```

```
     3    TOOLOW              OFF           0        TRUE
   10 = 3
     4    TOOLOW              OFF           1        TRUE
   12 = 4
     5    TOOLOW              OFF           2        TRUE
   13 = 5
     6    TOOLOW               ON           0       FALSE
    9 = 6
   11 = 6
     7    TOOLOW               ON           3       FALSE


GENERATED SCENARIOS: [FORMAT: STATE INPUT / OUTPUTS]

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

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

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

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

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

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

ST CONTAINS 13 nodes.

SCENARIO SEARCH TREE (13 total nodes):

Edge # (begin,  end)  Edge Label
------ ---------  ------------------------------------------------
     1 (  1,  6)  'WaterPres = 50 /
                         SafetyInjection = ON [P2] [R1] '
```

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

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

Total execution time: 0.000000 seconds
```

APPENDIX B

SAFETY INJECTION C/ATLAS SCENARIOS

B.1 Scenario 1

```
C      ***********************************************$
C      C/ATLAS TEST PROGRAM FOR SAFETY INJECTION SYSTEM $
C      ***********************************************$
C      $
 001000 BEGIN, ATLAS PROGRAM 'SAFETY INJECTION SYSTEM TEST 1' $
C      $
 001001 EXTEND, ATLAS NOUN, 'ABSTRACT_SIGNAL',
        FOR VERBS APPLY,
        SPEC 'NONE' $
 001002 EXTEND, ATLAS MODIFIERS FOR ABSTRACT_SIGNAL,
        'SIGNAL-TYPE',
           MOD-TYPE MNEMONIC-ONLY,
           USAGE STIMULUS-RESPONSE,
        'LEGAL-VALUES',
          USAGE STIMULUS-RESPONSE,
        'PRESENT-VALUE',
          USAGE STIMULUS $
 001003 EXTEND, ATLAS NOUN, 'DATALESS_EVENT' $
C      $
C      Define signals $
C      $
 002000 DEFINE, 'WATERPRESSURE', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE REAL, LEGAL-VALUES RANGE 0.0 TO 300.0,
        PRESENT-VALUE () $
 002001 DEFINE, 'BLOCK', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE DATALESS_EVENT $
```

```
 002002 DEFINE, 'RESET', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE ENUMERATED, LEGAL-VALUES {'ON', 'OFF'},
        PRESENT-VALUE () $
 002003 DEFINE, 'TREF', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE DATALESS_EVENT $
 002004 DEFINE, 'SAFETYINJECTION', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE ENUMERATED, LEGAL-VALUES {'ON', 'OFF'} $
C       $
C       Identify and enbable all needed events $
C       $
 003000 IDENTIFY, EVENT 'SI_OFF' AS 'SAFETYINJECTION' EQ 'OFF' $
 003001 IDENTIFY, EVENT 'SI_ON' AS 'SAFETYINJECTION' EQ 'ON' $
C       $
C       Set up initial input values $
C       $
 004000 APPLY, 'WATERPRESSURE', 150.0 $
 004001 APPLY, 'RESET', 'OFF', $
C       $
C       Enable monitoring of events $
C       $
 005000 ENABLE, EVENT 'SI_ON', 'SI_OFF' $
C       $
C       Start the test scenario $
C       $
 006000 WAIT FOR, 1 SEC $
 006001 APPLY, 'WATERPRESSURE', 50.0 $
 006002 WAIT FOR, EVENT 'SI_ON', MAX-TIME 5 SEC $
 006003 APPLY, 'WATERPRESSURE', 150.0 $
 006004 WAIT FOR, EVENT 'SI_OFF', MAX-TIME 5 SEC $
C       $
 999999 TERMINATE, ATLAS PROGRAM 'SAFETY INJECTION SYSTEM 1' $
```

B.2 Scenario 2

```
C      ***********************************************$
C      C/ATLAS TEST PROGRAM FOR SAFETY INJECTION SYSTEM $
C      ***********************************************$
C      $
 001000 BEGIN, ATLAS PROGRAM 'SAFETY INJECTION SYSTEM TEST 2' $
C      $
 001001 EXTEND, ATLAS NOUNS, 'ABSTRACT_SIGNAL',
        FOR VERBS APPLY,
        SPEC 'NONE' $
 001002 EXTEND, ATLAS MODIFIERS FOR ABSTRACT_SIGNAL,
        'SIGNAL-TYPE',
           MOD-TYPE MNEMONIC-ONLY,
           USAGE STIMULUS-RESPONSE,
        'LEGAL-VALUES',
          USAGE STIMULUS-RESPONSE,
        'PRESENT-VALUE',
          USAGE STIMULUS $
 001003 EXTEND, ATLAS NOUNS, 'DATALESS_EVENT' $
C      $
C      Define signals $
C      $
 002000 DEFINE, 'WATERPRESSURE', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE REAL, LEGAL-VALUES RANGE 0.0 TO 300.0,
        PREESENT-VALUE () $
 002001 DEFINE, 'BLOCK', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE DATALESS_EVENT $
 002002 DEFINE, 'RESET', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE ENUMERATED, LEGAL_VALUES {'ON', 'OFF'},
        PRESENT-VALUE () $
 002003 DEFINE, 'TREF', SIGNAL, ABSTRACT_SIGNAL,
```

```
        SIGNAL-TYPE DATALESS_EVENT $
 002004 DEFINE, 'SAFETYINJECTION', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE ENUMERATED, LEGAL-VALUES {'ON', 'OFF'} $
C      $
C      Identify and enbable all needed events $
C      $
 003000 IDENTIFY, EVENT 'SI_ON' AS 'SAFETYINJECTION' EQ 'ON' $
 003001 IDENTIFY, EVENT 'SI_OFF' AS 'SAFETYINJECTION' EQ 'OFF' $
C      $
C      Set up initial input values $
C      $
 004000 APPLY, 'WATERPRESSURE', 150.0 $
 004001 APPLY, 'RESET', 'OFF', $
C      $
C      Enable monitoring of events $
C      $
 005000 ENABLE, EVENT 'SI_ON', 'SI_OFF' $
C      $
C      Start the test scenario $
C      $
 006000 WAIT FOR, 1 SEC $
 006001 APPLY, 'WATERPRESSURE', 50.0 $
 006002 WAIT FOR, EVENT 'SI_ON', MAX-TIME 5 SEC $
 006003 APPLY, 'BLOCK' $
 006004 WAIT FOR, EVENT 'SI_OFF', MAX-TIME 3 SEC $
 006005 APPLY, 'TREF' $
 006006 APPLY, 'WATERPRESSURE', 150.0 $
 006007 WAIT FOR, 1 SEC $
 006008 APPLY, 'WATERPRESSURE', 50.0 $
 006009 WAIT FOR, EVENT 'SI_ON', MAX-TIME 5 SEC $
C      $
```

```
 999999 TERMINATE, ATLAS PROGRAM 'SAFETY INJECTION SYSTEM 1' $
```

B.3 Scenario 3

```
C      ********************************************$
C      C/ATLAS TEST PROGRAM FOR SAFETY INJECTION SYSTEM $
C      ********************************************$
C      $
 001000 BEGIN, ATLAS PROGRAM 'SAFETY INJECTION SYSTEM TEST 3' $
C      $
 001001 EXTEND, ATLAS NOUNS, 'ABSTRACT_SIGNAL',
        FOR VERBS APPLY $
 001002 EXTEND, ATLAS MODIFIERS FOR ABSTRACT_SIGNAL,
        'SIGNAL-TYPE',
           MOD-TYPE MNEMONIC-ONLY,
           USAGE STIMULUS-RESPONSE,
        'LEGAL-VALUES',
          USAGE STIMULUS-RESPONSE,
        'PRESENT-VALUE',
          USAGE STIMULUS $
 001003 EXTEND, ATLAS NOUNS, 'DATALESS_EVENT' $
C      $
C      Define signals $
C      $
 002000 DEFINE, 'WATERPRESSURE', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE REAL, LEGAL-VALUES RANGE 0.0 TO 300.0,
        PRESENT-VALUE () $
 002001 DEFINE, 'BLOCK', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE DATALESS_EVENT $
 002002 DEFINE, 'RESET', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE ENUMERATED, LEGAL_VALUES {'ON', 'OFF'},
        PRESENT-VALUE () $
```

```
 002003 DEFINE, 'TREF', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE DATALESS_EVENT $
 002004 DEFINE, 'SAFETYINJECTION', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE ENUMERATED, LEGAL-VALUES {'ON', 'OFF'} $
C      $
C      Identify and enbable all needed events $
C      $
 003000 IDENTIFY, EVENT 'SI_ON' AS 'SAFETYINJECTION' EQ 'ON' $
 003001 IDENTIFY, EVENT 'SI_OFF' AS 'SAFETYINJECTION' EQ 'OFF' $
C      $
C      Set up initial input values $
C      $
 004000 APPLY, 'WATERPRESSURE', 150.0 $
 004002 APPLY, 'RESET', 'OFF', $
C      $
C      Enable monitoring of events $
C      $
 005000 ENABLE, EVENT 'SI_ON', 'SI_OFF' $
C      $
C      Start the test scenario $
C      $
 006000 WAIT FOR, 1 SEC $
 006001 APPLY, 'WATERPRESSURE', 50.0 $
 006002 WAIT FOR, EVENT 'SI_ON', MAX-TIME 5 SEC $
 006003 APPLY, 'BLOCK' $
 006004 WAIT FOR, EVENT 'SI_OFF', MAX-TIME 3 SEC $
 006005 APPLY, 'TREF' $
 006006 WAIT FOR, 500 MSEC $
 006007 APPLY, 'TREF' $
 006008 WAIT FOR, 500 MSEC $
 006009 APPLY, 'TREF' $
```

```
 006010 WAIT FOR, EVENT 'SI_ON', MAX-TIME 3 SEC $
 006011 APPLY, 'BLOCK' $
 006012 WAIT FOR, EVENT 'SI_OFF', MAX-TIME 3 SEC $
C      $
 999999 TERMINATE, ATLAS PROGRAM 'SAFETY INJECTION SYSTEM 1' $
```

B.4 Scenario 4

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

```
        SIGNAL-TYPE DATALESS_EVENT $
 002002 DEFINE, 'RESET', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE ENUMERATED, LEGAL_VALUES {'ON', 'OFF'},
        PRESENT-VALUE () $
 002003 DEFINE, 'TREF', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE DATALESS_EVENT $
 002004 DEFINE, 'SAFETYINJECTION', SIGNAL, ABSTRACT_SIGNAL,
        SIGNAL-TYPE ENUMERATED, LEGAL-VALUES {'ON', 'OFF'} $
C       $
C       Identify and enbable all needed events $
C       $
 003000 IDENTIFY, EVENT 'SI_ON' AS 'SAFETYINJECTION' EQ 'ON' $
 003001 IDENTIFY, EVENT 'SI_OFF' AS 'SAFETYINJECTION' EQ 'OFF' $
C       $
C       Set up initial input values $
C       $
 004000 APPLY, 'WATERPRESSURE', 150.0 $
 004002 APPLY, 'RESET', 'OFF', $
C       $
C       Enable monitoring of events $
C       $
 005000 ENABLE, EVENT 'SI_ON', 'SI_OFF' $
C       $
C       Start the test scenario $
C       $
 006000 WAIT FOR, 1 SEC $
 006001 APPLY, 'WATERPRESSURE', 50.0 $
 006002 WAIT FOR, EVENT 'SI_ON', MAX-TIME 5 SEC $
 006003 APPLY, 'BLOCK' $
 006004 WAIT FOR, EVENT 'SI_OFF', MAX-TIME 3 SEC $
 006005 APPLY, 'RESET', 'ON' $
```

```
 006006 WAIT FOR, EVENT ’SI_ON’, MAX-TIME 3 SEC $
C      $
 999999 TERMINATE, ATLAS PROGRAM ’SAFETY INJECTION SYSTEM 1’ $
```

REFERENCES

[1] S. J. Cunning, *Automatic Test Generation for Discrete Event Oriented Real-Time Embedded Systems*. PhD thesis, University of Arizona, 2000.

[2] Y. Jaroch Gonzalez, "Automated validation of system requirements for embedded systems design," Master's thesis, University of Arizona, 1999.

[3] S. J. Cunning, T. C. Ewing, J. T. Olson, J. W. Rozenblit, and S. Schulz, "Towards an integrated, model-based codesign environment," in *Proceedings of the IEEE Conference and Workshop on the Engineering of Computer-Based Systems (ECBS '99)*, pp. 136–143, 1999.

[4] S. Schulz, J. W. Rozenblit, M. Mrva, and K. Buchenrieder, "Model-based codesign," *IEEE Computer*, vol. 31, no. 8, pp. 60–67, 1998.

[5] S. Schulz, *Model-Based Codesign for Real-Time Embedded Systems*. PhD thesis, University of Arizona, 2001.

[6] B. P. Zeigler, *Theory of modelling and simulation*. New York : Wiley, 1976.

[7] S. Schulz, T. C. Ewing, and J. W. Rozenblit, "Discrete event system specification (devs) and statemate statecharts equivalence for embedded systems modeling," in *Proceedings of the Seventh IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2000)*, pp. 308–316, 2000.

[8] *IEEE Std 716-1995 Standard Test Language for All Systems- Common/Abbreviated Test Language for All Systems (C/ATLAS)*. IEEE, 1995.

[9] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw, "Scr*: A toolset for specifying and analyzing requirements," in *COMPASS*, 1985.

[10] C. Heitmeyer, M. Archer, R. Bharadwaj, and R. Jeffords, "Tools for constructing requirements specifications: The scrtoolset at the age of ten," in *International Journal of Computer Systems Science and Engineering*, 2005.

[11] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "Automated consistency checking of requirements specifications," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 3, pp. 231–261, 1996.

[12] B. Dasarathy, "Timing constraints of real-time systems: Constructs for expressing them, methods of validating them," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 1, pp. 80–86, 1985.

[13] P. Bellini, R. Mattolini, and P. Nesi, "Temporal logics for real-time system specification," *ACM Computing Surveys*, vol. 32, no. 1, pp. 12–42, 2000.

[14] A. Pnueli, "Two decades of temporal logic: Achievements and challenges," *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pp. 78–78, 20-22 Oct 1997.

[15] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*. Springer, 1992.

[16] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, (Los Alamitos, CA), IEEE Computer Society Press, 1977.

[17] N. Malik, J. Baumgartner, S. Roberts, and R. Dobson, "A toolset for assisted formal verification," in *IEEE International Performance, Computing and Communications Conference(IPCCC '99)*, pp. 489–492, 1999.

[18] J. K. Deka, P. Dasgupta, and P. P. Chakrabarti, "An efficiently checkable subset of tctl for formal verification of transition systems with delays," in *Proceedings of the Twelfth International Conference On VLSI Design*, pp. 294–299, 1999.

[19] D. Drusinksy and M. T. Shing, "Creation and evaluation of formal specifications for system-of-systems development," in *IEEE International Conference on Systems, Man and Cybernetics*, vol. 2, pp. 1864–1869 Vol. 2, 2005.

[20] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of embedded systems: formal models, validation, and synthesis," *Proceedings of the IEEE*, vol. 85, no. 3, pp. 366–390, 1997.

[21] A. Krystosik, "Embedded systems modeling language," in *International Conference on Dependability of Computer Systems (DepCos-RELCOMEX '06)*, pp. 27–34, 2006.

[22] M. Frey and M. Podolsky, "Specifying and analysing distributed object-oriented systems," in *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, pp. 38–51, 1999.

[23] A. Fuxman, L. Liu, M. Pistore, M. Roveri, and J. Mylopoulos, "Specifying and analyzing early requirements: some experimental results," in *Proceedings of the 11th IEEE International Requirements Engineering Conference*, pp. 105–114, 2003.

[24] E. A. Lee, "What's ahead for embedded software?," *Computer*, vol. 33, no. 9, pp. 18–26, 2000.

[25] A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso, "Model checking early requirements specifications in tropos," in *Proceedingsof the Fifth IEEE International Symposium on Requirements Engineering*, pp. 174–181, 2001.

[26] S. Leue and G. Holzmann, "v-promela: a visual, object-oriented language for spin," in *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '99)*, pp. 14–23, 1999.

[27] O. Clarke, Edmund M. Jr.and Grumberg and D. A. Peled, *Model Checking*. MIT Press, 1999.

[28] G. J. Holzmann, *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, September 2003.

[29] J. Soukup, "Circuit layout," *Proceedings of the IEEE*, vol. 69, no. 10, pp. 1281–1304, 1981.

[30] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science, Number 4598, 13 May 1983*, vol. 220, 4598, pp. 671–680, 1983.

[31] B. Krishnamurthy, "An improved min-cut algorithm for partitioning vlsi networks," *IEEE Transactions on Computers*, vol. C-33, no. 5, pp. 438–446, 1984.

[32] M. C. McFarland, T. J. Kowalski, and T. J. Kowalski, "Incorporating bottom-up design into hardware synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 9, pp. 938–950, 1990.

[33] B. Kernighan and S. Lin, "An effective heuristic procedure for partitioning graphs," *The Bell System Technical Journal*, pp. 291–308, Feb 1970.

[34] C. J. Neill, P. A. Laplante, and P. A. Laplante, "Requirements engineering: the state of the practice," *IEEE Software*, vol. 20, no. 6, pp. 40–45, 2003.

[35] B. Berenbach and M. Gall, "Toward a unified model for requirements engineering," in *International Conference on Global Software Engineering (ICGSE '06)*, pp. 237–238, 2006.

[36] O. Djebbi, C. Salinesi, and G. Fanmuy, "Industry survey of product lines management tools: Requirements, qualities and open issues," in *15th IEEE International Requirements Engineering Conference (RE '07)*, pp. 301–306, 2007.

[37] P. J. Courtois and D. L. Parnas, "Documentation for safety critical software," in *Proceedings of the 15th International Conference on Software Engineering*, pp. 315–323, 1993.

[38] Y. Jaroch, S. Cunning, and J. Rozenblit, "Automated validation of system requirements for embedded systems design," in *Proceedings of the SCS Conference on AI, Simulation and Planning In High Autonomy Systems*, pp. 243–249, 2000.

[39] E. C.-H. Mak, "Automated testing using xml and devs," Master's thesis, University of Arizona, 2006.

[40] M. R. Garey and D. S. Johnson, *Computers and Intractability : A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, January 1979.

[41] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation (OSDI'04)*, (Berkeley, CA, USA), pp. 10–23, USENIX Association, 2004.