AUTOMATED VALIDATION OF SYSTEM REQUIREMENTS FOR EMBEDDED SYSTEMS DESIGN

Yarisa Jaroch, Steven Cunning, and Jerzy W. Rozenblit
Dept. of Electrical and Computer Engineering
The University of Arizona
Tucson, Arizona 85721-0104
e-mail: {yarisa/scunning/jr@ece.arizona.edu}

KEYWORDS: codesign, requirements testing, validation, embedded systems

1 ABSTRACT

Advancements in technology have increased the diversity and complexity of embedded systems. At the same time there is an increasing need for reducing the cost of embedded systems development and testing. The system requirements state what the customer wants the system to do. These requirements are the basis for planning a project. In the computer-based systems world, the requirements are analyzed and allocated to software and hardware. In many cases, either hardware, software, or various combinations can be used to meet the required performance and functionality. This is where other factors such as cost and flexibility influence the decision as to whether a requirement will be allocated to one or the other. The final design must meet the given requirements. If a tool is available to support the validation of the design against the systems requirements, then the time associated with the design process can be shortened. Our approach is to use model-based codesign to develop test scenarios based on the system requirements and use modeling and simulation to validate the design against the requirements. The approach will be demonstrated using an elevator controller model.

2 INTRODUCTION

Recognizing the need for better approaches and techniques for the design of embedded systems, the field of codesign has evolved (Buchenrieder and Rozenblit 1995); codesign encompasses hardware design and software design.

In the traditional design approach, the customer requirements are analyzed and used to develop the system specification. From the system specification, the requirements are partitioned into hardware and software requirements. Then, the hardware and the software are designed independently of each other. Upon completion of the design, the hardware and the software are brought back together during the integration phase. This approach is shown in Figure 1.

While in the traditional approach the hardware and the software are treated independently, in codesign the hardware and the software are viewed as a whole and are developed concurrently.

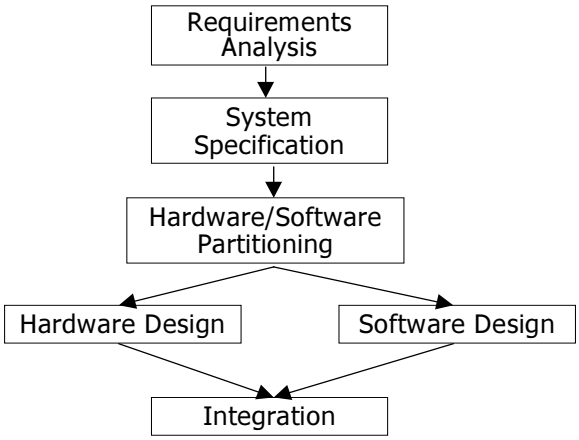

Figure 1. Traditional Design Approach

Codesign emphasizes the importance of modeling, simulation, and validation of the system before the hardware and the software are developed. These activities allow the developers to perform requirements allocation evaluations and trade-off analysis early in the process, and with knowledge of implications to the embedded system as a whole.

In model-based codesign (Cunning et al. 1999), the system requirements are used to design a model of the system. This model is then used for simulation. The output of the simulation is used to validate the model with respect to the requirements specifications, and to evaluate design alternatives. This approach can be depicted as shown in Figure 2.

In this area, automation of model validation can be of substantial help. Automatic model validation can

significantly reduce the time spent selecting the most suitable hardware/software requirements allocations.

As indicated before, the model validation is done through simulation. The simulation uses an experimental frame (Zeigler 1990). Typically the experimental frame is generated to test a particular requirement or related group of requirements. In order to validate a design model against all requirements, multiple experimental frames must be generated. The experimental frame developed to support model validation for model-based codesign is generic. The uniqueness of the experimental frame is encapsulated by the test cases defined and by the input/output modules. Since the experimental frame is generic, the same frame can be used with multiple test cases to validate the model thus reducing the number of frames required.

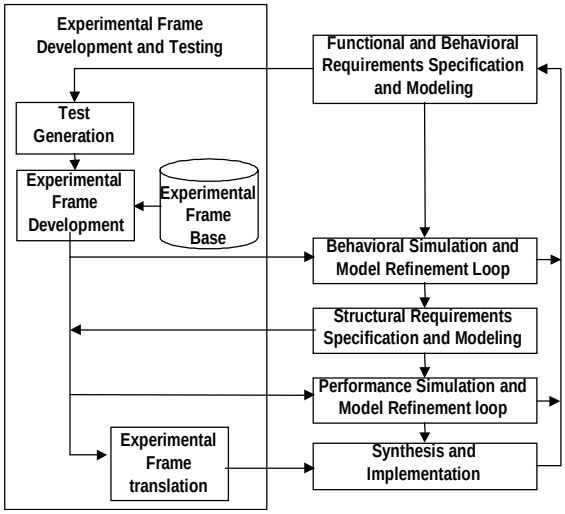

Figure 2. Model-Based Codesign Approach

3 VALIDATION OF REQUIREMENTS

Because the requirements are the basis for the project, systems are tested to ensure requirements compliance. Researchers are working towards developing a method for test development using the requirements. For this to work, requirements must be unambiguous. Currently many companies specify their requirements using natural languages. However, it is very difficult to capture requirements that are not open to interpretation using natural language. This is why requirements notations such as RLP (Requirements Language Processor) (Davis 1982), PVS (Prototype Verification System) (Butler 1996) and SCR (Software Cost Reduction) (Heitmeyer et al. 1996) have been developed.

Although many formal notations have been developed, not one has caught on. Systems engineers and customers prefer to maintain their requirements in a format that is easily understood. Yet, formal notations can lead to early detection of missing requirements and misunderstandings.

We propose a middle ground: the customer specifies the requirements in natural language and the systems engineer or the designer translates the requirements into a formal requirement notation (Cunning and Rozenblit 1999). Our goal is to obtain the benefits of both natural languages and formal requirement notations without redefining the existing requirements capture methods. Because the requirements are precisely defined in a mathematical language, the requirements can be used to develop test cases for behavioral models of the system.

Our experimental frame development and testing approach consists of several steps as shown in Figure 3. First, the requirements are received by the customer in English sentences and manually translated into the requirements model. The requirements model uses SCR to capture the requirements. The step of translating the requirements from English sentences into SCR uncovers missing and incomplete requirements. Once the requirements model is complete, the scenario generation algorithm is used to develop test scenarios from the requirements model. The test scenarios are then translated into ATLAS test programs (See section 0). Using the test programs, experimental frames are synthesized. The experimental frames are then used to apply stimuli to the design model and to gather the design model responses. The results of the simulation are stored for further analysis.

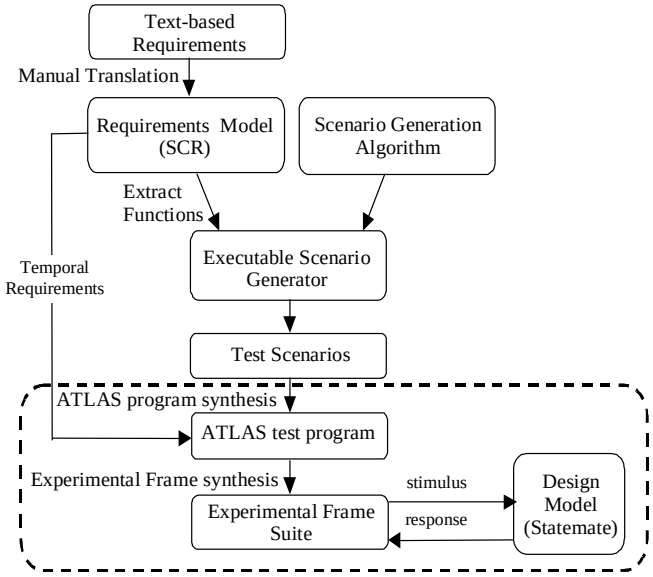

Figure 3. Test Scenario Generation and Use

Automation of the steps encompassed by the dotted line in Figure 3 represent the focus of this paper. These steps will be described in detail in the remaining sections.

4 MODELING AND SIMULATION

As stated earlier, simulation is used to validate the design models with respect to the requirements specifications, and to evaluate design alternatives.

An experimental frame is a representation of the environments in which the model resides (Zeigler 1990). The frame subjects the model to the environment input stimuli. It records the model responses to the stimuli and collects data about such responses. Finally, it controls the experimentation by placing constraints on the model and by monitoring these constraints.

In practice, the experimental frames consist of a generator, a transducer and an acceptor. The generator produces the environment stimuli for the model. The transducer calculates metrics such as throughput and turnaround time. The acceptor controls the simulation.

Figure 4 shows a graphical view of a simulation using experimental frames. Simulation using experimental frames allows for different conditions of the environment to be represented. Also, the same experimental frame can be used for simulation of a different model of the same system (assuming the model inputs and outputs are unchanged). This last feature is very important in modeling were developers might want to evaluate multiple designs to determine which one is best suited.

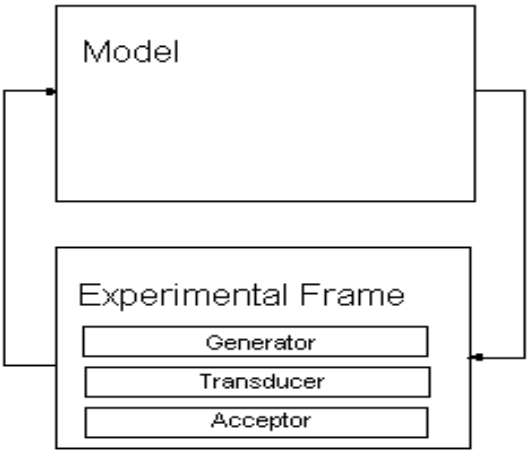

Figure 4. Simulation using an Experimental Frame

5 ATLAS

ATLAS, or C/ATLAS – Common/Abbreviated Test Language for All Systems, is the language chosen to communicate the test scenarios. Choosing an existing language saves the time of developing a practical and unambiguous language. Moreover the use of a standard language provides the opportunity for additional uses of the developed test cases. ATLAS is maintained by IEEE Standard 716-1995. For background information on ATLAS see IEEE ATLAS 2000 interest group 1997.

The ATLAS procedural statements of interest provides two statements: APPLY and WAIT FOR. The APPLY statement indicates which signals should be set and the value they should be set to. The WAIT FOR has two variations: WAIT FOR time and WAIT FOR EVENT event MAX-TIME time. The first one states the time to wait before performing the next statement while the second one is used to wait for an event. The time associated with the second WAIT FOR is used to specify the maximum amount of time to wait for the specified event. An example of the APPLY and WAIT FOR statements follow.

```
C    $
C    Set up initial input values $
C    $
 000301 APPLY, 'BLOCK', 'OFF' $
 000302 APPLY, 'RESET', 'OFF' $
 000303 APPLY, 'TREF', 'FALSE' $
C    $
C    Start the test scenario $
C    $
 000401 WAIT FOR 1 sec $
 000402 APPLY, 'WATERPRESSURE', 50.0 $
 000403 WAIT FOR EVENT, 'SI_ON', MAX-TIME 5 sec $
 000404 APPLY, 'WATERPRESSURE', 150.0 $
 000405 APPLY, 'TESTPRESSURE', 'UP' $
```

6 EXPERIMENTAL FRAME TOOL SET

The experimental frame tool set was developed for the Statemate modeling tool (Harel 1990). Statemate uses state charts to design the system and it supports complex reactive systems. This tool was chosen because of it's modeling and simulation capabilities, and because of the popularity of Statemate in industry.

The experimental frame tool set approach begins with the design of the system model based on the system requirements. After the design is complete, the input and output information is gathered from Statemate and saved into the data dictionary file. The data dictionary file and the ATLAS test scenario file are then used to generate the experimental frames input/output modules and the simulation input file, a translated version of the test scenarios. When the input/output modules have been generated, the input/output modules and the model code are

used to couple the model and the experimental frame for the simulation. The simulation is then executed using the simulation input file. During simulation, the input/output changes are saved.

The experimental frame tool set approach is shown in Figure 5. The tools are mapped to the approach as follows: the data dictionary tool implements the "Get I/O Info. from Statemate", the preprocess tool implements the "Generate I/O modules for EF", and the experimental frame implements the "Setup EF for simulation".

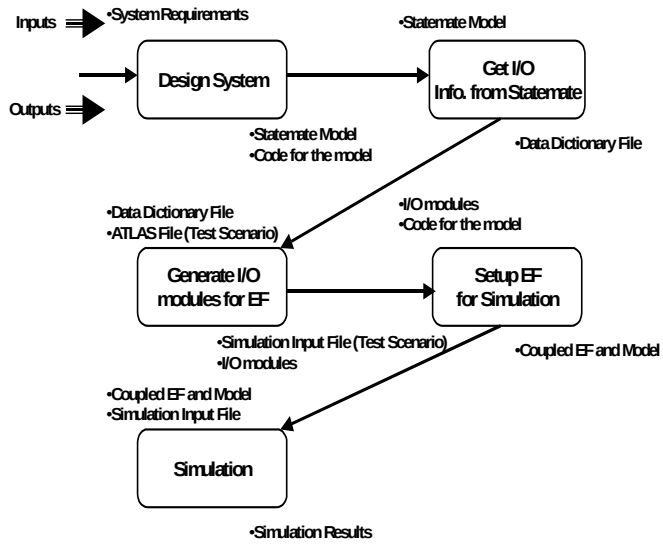

Figure 5. Experimental Frame Tool Set Approach

As stated before, the test scenarios are documented using ATLAS. In selecting ATLAS, time expressions were evaluated to determine which timing constraints could be represented. Dasarathy (Dasarathy 1985) classifies timing constraints into:

Maximum - No more than t amount of time may elapse between the occurrence of one event and the occurrence of another.
Minimum - No less than t amount of time must elapse between these two events.
Durational - An event must occur for t amount of time.

There are four kinds of maximum and minimum timing constraints. The constraints are:

S-S Combination - time between the occurrence of two stimuli.
S-R Combination - time between the arrival of a stimuli and system response
R-S Combination - time between the system response and the next stimulus from the environment.
R-R Combination - time between two responses.

The current work supports S-S and R-S timing constraints dynamically while S-R and R-R timing constraints are supported through post simulation analysis. S-S and R-S are constraints on the input stimuli (i.e. the environment) while the S-R and R-R are constraints on the response from the system.

6.1 Data Dictionary Tool

As mentioned in the previous section, one of the inputs to the experimental frame is the Statemate model. Before simulation, the Statemate model is used to generate a list of all the inputs and outputs, and their types. The advantage of this method is that it guarantees the experimental frame input/output modules will interface correctly with the design model.

In Statemate the inputs and outputs are defined in the data dictionary. Although Statemate provides a report generation tool which provides the contents of the data dictionary, the tool is available only from within the graphical user interface. Since, the goal of this work is to be able to automate the experimental frame synthesis, the tool provided was not ideal. For this reason a new tool was developed using the Statemate dataport feature. The dataport feature provides a library of C routines, which can be used to access the Statemate data for a particular model. The tool output follows the same format as Statemate's report generation tool when the report generation tool is set to ASCII.

The tool output looks as follows:

```
SIGNALNAME
Defined in chart: CHART
Type: Data-item
Usage: UsageType
-------------------------------------------------
```

6.2 Preprocessing Tool

The preprocessing tool serves a dual purpose. First, it generates the input and output modules used with the generic experimental frame. Second, it replaces inputs and outputs name strings in the test file with numbers. The purpose of replacing strings with numbers is to decrease the execution time of the simulation since string comparisons are much more time consuming than integer comparisons.

The inputs to the preprocessing tool are the ATLAS test scenario file and the file generated with the data dictionary tool. The outputs are the preprocessed test file and the input and output modules of the experimental frame.

When the preprocess tool is executed, it first reads the data dictionary output file, and then reads the ATLAS test scenario file. While the tool parses the ATLAS file, it saves the definition of all the events. It also creates a new file

called preprocessed, which is the file the experimental frame expects. In this file all strings have been replaced. Before adding a statement to change an input, the preprocess tool verifies that the input is defined in Statemate. If it is not defined in Statemate, it issues an error message. When the preprocess tool finds a statement commanding the generator to wait for a particular event, it finds the definition of the event and it adds it to the preprocessed file. The preprocess tool also verifies that the signal related to the event and enumerations in use are in Statemate. Similar to the input and output signals, the enumeration types inside the ATLAS file are indexed to remove string comparisons.

After the ATLAS file has been parsed, the preprocess tool proceeds to generate the input and output modules of the experimental frame. These modules are contained within the files io.h and add_to_cbk.h. The contents and structure of these files are explained in the next section.

6.3 Experimental Frame

The experimental frame developed here connects to the design model code generated by Statemate for a model using the hooks put in place by Statemate. The code generated by Statemate provides a series of user files where code can be added to supplement the Statemate code. The advantage of this is that the model as defined within Statemate does not have to be modified to support simulation with the external experimental frame. Also, because the experimental frame code is not physically linked to the model, the same code can be used with different design models.

While studying Statemate with the idea of a generic experimental frame, we found that in Statemate, a different function call must be used for each type of signal. In order to keep the experimental frame generic and model independent, the concept of I/O modules was developed and applied. The I/O modules encapsulate the uniqueness of the model but provide a common interface to the generator and the transducer. The generator would then call set_input from I/O for any input and I/O would determine the type of the input and call the appropriate Statemate routine. The file io.h generated by preprocessed contains the I/O modules.

The file io.h contains four types of routines: set_input, print_xxx, setup_initial_values and enum_map. Print_xxx is not one but many routines. One for each input and output from the model. They encapsulate the type of each signal in order to write it's value to the simulation output file. Setup_initial_values initializes the test scenario before the simulation begins. The information for setting the initial values is read from the ATLAS file.

Because, the time needed for string comparisons is longer than the time for integer comparisons all the signals and the enumerations were coded. Set_input contains a mapping from the coded input to the actual Statemate input. Similarly, enum_map converts the coded enumeration into the appropriate enumeration value.

6.3.1 Generator

The generator applies stimuli to the model. The generator reads the preprocessed file and determines whether the statement read is a WAIT FOR or APPLY. If the statement is an APPLY, the generator invokes set_input. If the statement is a WAIT FOR time, the generator waits until the time has elapsed before reading and executing the next statement. If the statement is a WAIT FOR event MAX-TIME time, the generator waits until the event has occurred or the time has elapsed before reading and executing the next statement.

To wait for an event, the generator uses the Statemate callback feature. A callback binds a procedure to a signal. Every time the signal changes value the routine gets executed. This routine sets a flag the generator can see when the flag is set, the generator compares the value of the signal against the event definition. If the criteria for the event is met or the time indicated has elapsed, the generator proceeds to the next statement.

6.3.2 Acceptor

The acceptor controls the simulation. In essence the simulation control is based on the contents of the test scenario file. The file indicates when to apply the stimuli and when to wait. The simulation is terminated when the last statement in the preprocessed file has been executed. Therefore, is recommended that there is a WAIT FOR statement at the end of the test scenario. Ending on a WAIT FOR with an appropriate duration allows the simulation to continue long enough track changes to the system based on the last APPLY statement.

6.3.3 Transducer

The transducer records the model's stimuli from the generator and records the models output response. The data is time stamped and saved in the simulation output file. The transducer concept is implemented by a combination of procedures, the first being set_input. Every time set_input is called, it writes out which signal is being set and the value it is being set to. The transducer also relies on the Statemate callback feature. Each of the print_xxx routines is bound to the appropriate signal. Every time a signal changes value, the appropriate print_xxx routine is called, recording the signal name, the value it changed to, and the time it changed. The Statemate callback is initialized using the routine cbk_init contained in the add_to_cbk.h file.

7 EXPERIMENTAL RESULTS

The approach described in this document was tested using an elevator model. The following sections describe the model and the results after the execution of the simulations.

The elevator model represents a simple elevator in a three story building. At the first floor there is a button to signal when someone wants to go up. At the second floor there are two buttons, one to indicate someone wants to go up and the other one to indicate someone wants to go down. The third floor has one button to indicate someone wants to go down. Inside the elevator there are three buttons, one for each floor. Also, the elevator has sensors to detect when it arrives to a floor.

For the test scenario, the elevator is assumed to be stationary in the first floor and no one has pressed an elevator button. When the simulation starts, someone on the second floor requests the elevator to go up. Shortly after the elevator is going up, a new request is received from the first floor. When the elevator doors open, the person climbs into the elevator and request the third floor.

A portion of the ATLAS file for the scenario is shown below.

```
...
C    $
C    Start the test scenario $
C    $
 000401 WAIT FOR, 0 sec $
 000402 APPLY, 'FLOOR_REQUESTS.FLOOR_2_UP', 1 $
 000403 WAIT FOR EVENT, 'GOING_UP', MAX-TIME 2 sec $
 000404 APPLY, 'FLOOR_REQUESTS.FLOOR_1_UP', 1 $
 000405 WAIT FOR EVENT, 'AT_2_FLOOR', MAX-TIME 5 SEC $
 000406 WAIT FOR EVENT, 'DOOR_OPENED', MAX-TIME 2.2 sec $
 000407 APPLY, 'RIDER_REQUESTS.FLOOR_3_REQUEST', 1 $
 000408 WAIT FOR EVENT, 'AT_1_FLOOR', MAX-TIME 20 sec $
 000409 WAIT FOR EVENT, 'STOPPED', MAX-TIME 2 sec $
```

A sample portion of the data dictionary tool output follows.

```
DIRECTION
Defined in chart: ELEVATOR
Definition: {UP,DOWN,STATIONARY}
Data type: Enum-type
--------------------------------------------------------------------------------

CUR_FLOOR
Defined in chart: ELEVATOR_ACTIVITIES
Usage: Variable
Data type: Integer min=1 max=3
--------------------------------------------------------------------------------

CURRENT_DIRECTION
Defined in chart: ELEVATOR_CONTROL
Usage: Variable
Data type: DIRECTION
--------------------------------------------------------------------------------
```

The final simulation output is shown below:

```
Begin Simulation @ 0.703660
DELAY_TIME = 7 @ 0.704757
CURRENT_DIRECTION = 2 @ 0.704850
CUR_FLOOR = 1 @ 0.704872
-> FLOOR_REQUESTS.FLOOR_2_UP 1 @ 0.705744
FLOOR_REQUESTS.FLOOR_2_UP = 1.000000 @ 0.706162
GO_UP = 1.000000 @ 0.706298
CURRENT_DIRECTION = 0 @ 0.706332
GOING_UP = 1.000000 @ 0.706605
-> FLOOR_REQUESTS.FLOOR_1_UP 1 @ 0.706850
FLOOR_REQUESTS.FLOOR_1_UP = 1.000000 @ 0.707035
AT_FLOOR = 1.000000 @ 3.706632
CUR_FLOOR = 2 @ 3.706783
FLOOR_REQUESTS.FLOOR_2_UP = 0.000000 @ 3.707107
GOING_UP = 0.000000 @ 3.707144
DOORS_OPEN = 1.000000 @ 3.707540
-> RIDER_REQUESTS.FLOOR_3_REQUEST 1 @ 3.707999
RIDER_REQUESTS.FLOOR_3_REQUEST = 1.000000 @ 3.708435
GO_UP = 1.000000 @ 3.708585
DOORS_OPEN = 0.000000 @ 3.708767
GOING_UP = 1.000000 @ 3.708794
AT_FLOOR = 1.000000 @ 6.708878
CUR_FLOOR = 3 @ 6.709041
RIDER_REQUESTS.FLOOR_3_REQUEST = 0.000000 @ 6.709160
GOING_UP = 0.000000 @ 6.709189
CURRENT_DIRECTION = 1 @ 6.709235
DOORS_OPEN = 1.000000 @ 6.709400
DELAY_TIME = 6 @ 7.709719
DELAY_TIME = 5 @ 8.710001
DELAY_TIME = 4 @ 9.710282
DELAY_TIME = 3 @ 10.710586
DELAY_TIME = 2 @ 11.710890
DELAY_TIME = 1 @ 12.711199
DELAY_TIME = 0 @ 13.711507
DOORS_OPEN = 0.000000 @ 14.711752
CURRENT_DIRECTION = 2 @ 14.711782
GO_DOWN = 1.000000 @ 14.711893
CURRENT_DIRECTION = 1 @ 14.711927
GOING_DOWN = 1.000000 @ 14.712093
AT_FLOOR = 1.000000 @ 17.712219
CUR_FLOOR = 2 @ 17.712366
GO_DOWN = 1.000000 @ 17.712482
AT_FLOOR = 1.000000 @ 20.712736
CUR_FLOOR = 1 @ 20.712885
FLOOR_REQUESTS.FLOOR_1_UP = 0.000000 @ 20.713247
GOING_DOWN = 0.000000 @ 20.713284
CURRENT_DIRECTION = 0 @ 20.713329
DELAY_TIME = 7 @ 20.713785
DOORS_OPEN = 1.000000 @ 20.713818
CURRENT_DIRECTION = 2 @ 20.713846
End of Simulation Reached @ 20.715369
```

The output of the scenario shows the responses of the elevator to the floor requests in the test scenario. It also shows the delay time associated with the elevator and the current position of the elevator in the simulation.

8 CONCLUSIONS

The approach and the tools developed here demonstrate automatic validation for embedded systems is possible. This approach provides a new technique for the design of embedded systems, and can save the industry significant

time and money since many hardware/software incompatibilities can be discovered early in the design cycle.

Although model-based codesign is the foundation for this work, the approach demonstrated can be applied not only to model-based codesign but also to any model-based method (Evans 1995). However, in order for this approach to work, test scenarios must be developed in an unambiguous language such as ATLAS.

Even though the work here uses Statemate, the same ideas can be applied to other modeling environments. Statemate was chosen because of it's modeling and simulation capabilities in order to prove the viability of our approach.

REFERENCES

Buchenrieder, K. and J. W. Rozenblit. 1995. "Codesign: An Overview", in Codesign: Computer-Aided Software Hardware Engineering, J. W. Rozenblit and K. Buchenrieder, Eds., IEEE Press, New York, 1-15.

Butler, R. W. 1996. "An Introduction to Requirements Capture Using PVS: Specification of a Simple Autopilot." May 1996, National Aeronautics and Space Administration, Langley Research Center, Hampton, VA.

Cunning, S. J. and J. W. Rozenblit. 1999. "Test Scenario Generation from a Structured Requirements Specification." *Proceedings of the 1999 IEEE Conference and Workshop on Engineering of Computer Based Systems (ECBS'99)*, Nashville, TN, March 7-12, 166-172.

Cunning, S. J.; T. C. Ewing; J. T. Olson; J. W. Rozenblit; S. Schulz. 1999. "Towards an Integrated, Model-Based Codesign Environment." *Proceedings of the IEEE Conference and Workshop on Engineering of Computer Based Systems (ECBS '99)*, Nashville, TN, March 7-12, 136-143.

Dasarathy, B. 1985. "Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them." *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 1 (January): 80-86.

Davis, A. 1982. "The Design of a Family of Application-Oriented Requirements Languages." *IEEE Computer* 15 (May), 21-28.

Evans, D. G. and D. Morris. 1995. "Applying Modeling to Embedded Computer Systems Design." In Codesign: Computer-Aided Software Hardware Engineering, J. W. Rozenblit and K. Buchenrieder, Eds., IEEE Press, New York, 98-116.

Harel, D. 1990. "STATEMATE: A Working Environment for the Development of Complex Reactive Systems." *IEEE Transactions on Software Engineering*, 16(4), 403-414.

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

Zeigler, B. P. 1990. "Object-Oriented Simulation with Hierarchical, Modular Models Intelligent Agents and Endomorphic Systems", Academic Press, Inc., California.

IEEE ATLAS 2000 interest group. 1997. "ATLAS 2000 Introductory Guide." Rev. B, 13 February.