Specification-based Prototyping for Embedded Systems*

Jeffrey M. Thompson[1], Mats P.E. Heimdahl[1], and Steven P. Miller[2]

[1] Computer Science and Engineering Department, University of Minnesota
Minneapolis, MN 55455,
{thompson, heimdahl}@cs.umn.edu

[2] Advanced Technology Center, Rockwell Collins
Cedar Rapids, IA 52402
spmiller@collins.rockwell.com

Abstract. Specification of software for safety critical, embedded computer systems has been widely addressed in literature. To achieve the high level of confidence in a specification's correctness necessary in many applications, manual inspections, formal verification, and simulation must be used in concert. Researchers have successfully addressed issues in inspection and verification; however, results in the areas of execution and simulation of specifications have not made as large an impact as desired.

In this paper we present an approach to *specification-based prototyping* which addresses this issue. It combines the advantages of rigorous formal specifications and rapid systems prototyping. The approach lets us refine a formal executable model of the system requirements to a detailed model of the software requirements. Throughout this refinement process, the specification is used as a prototype of the proposed software. Thus, we guarantee that the formal specification of the system is always consistent with the observed behavior of the prototype. The approach is supported with the NIMBUS environment, a framework that allows the formal specification to execute while interacting with software models of its embedding environment or even the physical environment itself (hardware-in-the-loop simulation).

1 Introduction

Validating software specifications for embedded systems presents particularly difficult problems. The software's correctness cannot be determined without considering its intended operating environment. In these systems, the software must interact with a variety of analog and digital components and be able to detect and recover from error conditions in the environment. In addition, the software is often subject to rigorous safety and performance constraints.

Assurance that the software specification possesses desired properties can be achieved through (1) manual inspections, (2) formal verification, or (3) simulation and testing. To achieve the high level of confidence in the correctness required in many of today's critical embedded systems, all three approaches must

* This work has been partially supported by NSF grants CCR-9624324 and CCR-9615088.

be used in concert. This paper addresses the some of the capabilities of NIMBUS, our environment for embedded systems development.

The capability to dynamically analyze, or execute, the description of a software system early in a project has many advantages: it helps the analyst to evaluate and address poorly understood aspects of a design, improves communication between the different parties involved in development, allows empirical evaluation of design alternatives, and is one of the more feasible ways of validating a system's behavior.

Rapid prototyping is one way of achieving executability for systems. It has been successful when specialized tools and languages are applied to specific, well defined domains. For example, prototyping of user interfaces and database systems has been highly successful. Languages in these domains, such as PowerBuilder and Visual Basic, provide powerful, high-level features and have achieved great success in industry. In the embedded systems domain, however, rapid prototyping does not appear to have been as successful.

In this paper, we focus on an approach to simulation and debugging of formal specifications for embedded systems called *specification-based prototyping* [4]. Within the context of specification execution and simulation, specification-based prototyping combines the advantages of traditional formal specifications (e.g., unambiguity and analyzability) with the advantages of rapid prototyping (e.g., risk management and early end-user involvement). The approach lets us refine a formal executable model of the system requirements to a detailed model of the software requirements. Throughout this refinement process, the specification is used as an early prototype of the proposed software. By using the specification as the prototype, most of the problems that plague traditional code-based prototyping disappear. First, the formal specification will always be consistent with the behavior of the prototype (excluding real-time response) and the specification is, by definition, updated as the prototype evolves. Second, the capability to evolve the prototype into a production system is largely eliminated. Finally, the dynamic evaluation of the prototype can be augmented with formal analysis.

To enable specification-based prototyping, we have developed the NIMBUS requirements engineering environment. NIMBUS, among other things, allows us to dynamically evaluate an RSML (Requirements State Machine Language) specification while interacting with (1) user input or text file input scripts, (2) RSML models of the components in the embedding environment, (3) software simulations of the components, or (4) the physical components themselves. When starting to develop NIMBUS, we identified the following fundamental properties such an environment must posses. First, it must support the execution of the specification while interacting with accurate models of the components in the surrounding environment, be that RSML specifications, numerical simulations, statistical models, or physical hardware. Second, the environment must allow an analyst to easily modify and interchange the models of the components. Third, as the specification is being refined to a design and finally production code, there should not be any large conceptual leaps in the way in which the control software communicates with the embedding environment.

In the next section, we provide a short overview of some related approaches to prototyping and executable specifications. Section 3 presents a small example and Section 4 outlines RSML and the NIMBUS environment. In Section 5 we discuss how NIMBUS is used to evaluate and refine a systems requirements model to a software requirements model. Section 6 concludes.

2 Related Work

2.1 Executable Specification Languages

An executable specification language is a formally well defined, very high-level specialized programming language. Most executable specification languages are intended to play many roles in the software development process. For instance, languages such as PAISLey [26], ASLAN [3], and REFINE [1] are intended to replace requirements specifications, design specifications, and, in some instances, implementation code. Executable specification languages have achieved some success and have been applied to industrial size projects. Many languages have elaborate tool support and facilitate refinement of a high level specification into more detailed design descriptions or implementation code.

Nevertheless, current executable specification languages have several drawbacks. Most importantly, the syntax and semantics are close to traditional programming languages. Therefore, currently they do not provide the level of abstraction and readability necessary for a requirements notation [8, 9].

Notable exceptions to the languages discussed above are a collection of state-based notations. Statecharts [10, 11], SCR (Software Cost Reduction) [15, 16], and the RSML [19], are very high-level and provide excellent support for inspections since they are relatively easy to use and understand for all stake holders in a specification effort. These languages allow automated verification of properties such as completeness and consistency [12, 15], and efforts are underway to model check state-based specifications of large software systems [2, 5].

Even so, none of the tools supporting these languages met the requirements that we established for a prototyping environment. SCR* and the original RSML tool did not allow as flexible nor as easy an integration of component models as we desired. Statemate provides the ability to integrate with other tools; however, this integration is achieved via a complex simulation backplane. This differs from the goals that we had for the NIMBUS environment: we wanted a framework in which it would be easy to integrate many different models of the components in the environment and which used a simple model of inter-component interaction, not a complex co-simulation tool.

2.2 Prototyping

There are two main approaches to prototyping. One approach is to develop a draft implementation to learn more about the requirements, throw the prototype away, and then develop production quality code based on the experiences from the prototyping effort. The other approach is to develop a high quality system from the start and then evolve the prototype over time. Unfortunately, there are problems with both approaches.

The most common problem with throw away prototyping is managerial, many projects start developing a throw away prototype that is later, in a futile attempt to save time, evolved and delivered as a production system. This misuse of a throw-away prototype inevitably leads to unstructured and difficult to maintain systems.

Dedicated prototyping languages have been developed to support evolutionary prototyping [18, 23]. These languages simplify the prototyping effort by supporting execution of partial models and providing default behavior for under-specified parts of the software. Although prototyping languages have achieved some initial success, it is not clear that they provide significant advantages over

traditional high-level programming languages. Evolutionary prototyping often lead to unstructured and difficult to maintain systems. Furthermore, incremental changes to the prototype may not be captured in the requirements specification and design documentation which leads to inconsistent documentation and a maintenance nightmare.

Software prototypes have been successfully used for certain classes of systems, for example, human-machine interfaces and information systems. However, their success in embedded systems development has been limited [6]. Clearly, a discussion of every other prototyping technique is beyond the scope of this paper. Nevertheless, most work in prototyping is, in our opinion, too close to design and implementation or is not suitable to the problem domain of embedded safety-critical systems.

Notable examples of work in prototyping include PSDL [18, 22] and Rapide [20, 21]. PSDL is based on having a reusable library of Ada modules which can be used to animate the prototype. Nevertheless, it seems that this approach would preclude execution until a fairly detailed specification was developed. Rapide is a useful prototyping system, but it does not have the capability to integrate as easily with other tools that desired. In addition, Rapide's scope is too broad for our needs; we wanted a tool-set that was focused on the challenges presented by embedded systems.

3 The Altitude Switch

This section describes a simple example drawn from the the avionics domain: the Altitude Switch (ASW). The ASW is a hypothetical device that turns power on to another subsystem when the aircraft descends below a threshold altitude. While the ASW appears almost trivial, it raises a surprising number of issues, particularly regarding how it interacts with its environment.

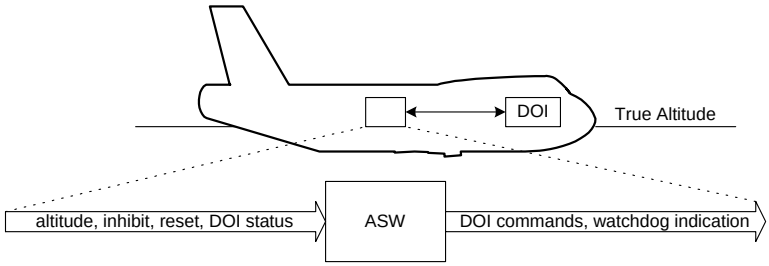

Fig. 1. The ASW system in its environment

The ASW and its environment are shown in Figure 1. The ASW receives altitude information from an analog radio altimeter and two digital radio altimeters, with the altitude taken as the lowest valid altitude seen. If the altitude cannot be determined for more than two seconds, the ASW indicates a fault by failing to strobe a watchdog timer. A fault is also indicated if internal failures are detected in the ASW. The detection of a fault turns on an indicator lamp within the cockpit.

The environment in which the ASW operates includes several features which make the system more interesting. First, the ASW software does not have complete control over the DOI. The DOI can be turned on or off at any time by other devices on the aircraft. Second, the functioning of the ASW can be inhibited or reset at any time. This raises questions, for example, about how the ASW should operate if it is reset while below the threshold altitude. Finally, the ASW must interface with three different altimeters; furthermore, the analog and digital altimeters are significantly different in terms of the information that they provide.

The next section introduces the RSML language by providing an overview of the high-level ASW requirements specification.

4 RSML and the NIMBUS Environment

RSML was developed as a requirements specification language for embedded systems and is based on David Harel's Statecharts [10]. One of the main design goals of RSML was readability and understandability by non-computer professionals such as users, engineers in the application domain, managers, and representatives from regulatory agencies [19].

4.1 Introduction to RSML

An RSML specification consists of a collection of *variables*, *states*, *transitions*, *functions*, *macros*, *constants*, and *interfaces*.

Variables in the specification allow the analyst to record the values reported by various external sensors (in the case of input variable) and provide a place to capture the values of the outputs of the system prior to sending them out in a message (in the case of output variables). Figure 2 shows the input variable definition for the Altitude in the ASW requirements.

InVariable Altitude : **Numeric**
Units : ft
Initial Value : 0
Expected Min : 0
Expected Max : 40,000

Fig. 2. An Input Variable definition from the ASW requirements

States are organized in a hierarchical fashion as in Statecharts. RSML includes three different types of states – *compound* states, *parallel* states, and *atomic* states. Atomic states are analogous to those in traditional finite state machines. Parallel states are used to represent the inherently parallel or concurrent parts of the system being modeled. Finally, compound states are used both to hide the detail of certain parts of the state machine so as to make the resulting model easier to comprehend and to encapsulate certain behaviors in the machine.

The state hierarchy modeling the high level ASW requirements could be represented as in Figure 3. This representation includes all three types of states. *FullyOperational* is a parallel state with three direct children. All of these are compound states which contain only atomic states (*Above*, *Below*, etc.).

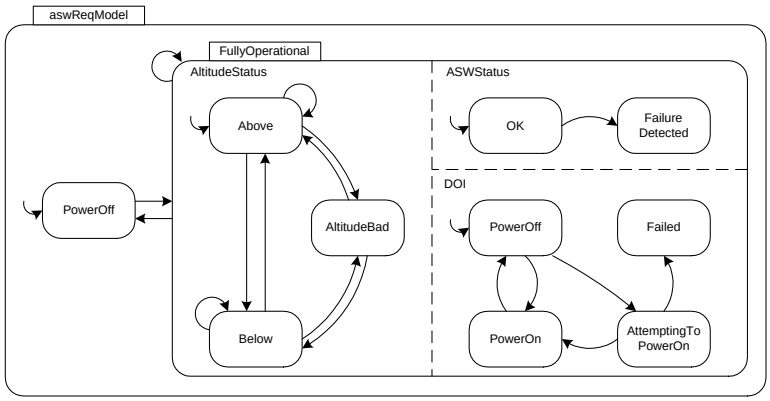

Fig. 3. High-level ASW Model

Transitions in RSML control the way in which the state machine can move from one state to another. A transition consists of a source state, a destination state, a trigger event, a guarding condition, and a set of action events that is produced when the transition is taken. In order to take an RSML transition, the following must be true: (1) the source state must be currently active, (2) the trigger event must occur while the source state is active, and (3) when the trigger event occurs, the guarding condition must evaluate to true. If all of these conditions are satisfied then the destination state will become active, the source state will become inactive, and the action events will be produced.

The guarding condition is simply a predicate logic expression over the various states and variables in the specification; however, during the TCAS project, the team that developed RSML (the Irvine Safety Research Group led by Dr. Nancy Leveson) discovered that the guarding conditions required to accurately capture the requirements were often complex. The prepositional logic notation traditionally used to define these conditions did not scale well to complex expressions and quickly became unreadable. To overcome this problem, they decided to use a tabular representation of disjunctive normal form (DNF) that they called AND/OR tables. The far-left column of the AND/OR table lists the logical phrases. Each of the other columns is a conjunction of those phrases and contains the logical values of the expressions. If one of the columns is true, then the table evaluates to true. A column evaluates to true if all of its elements match the truth values of the associated predicates. A dot denotes "don't care." Figure 4 (a) shows a transition from the high-level ASW requirements.

To further increase the readability of the specification, the Irvine Group introduced many other syntactic conventions in RSML. For example, they allow expressions used in the predicates to be defined as simple case-statement *functions* and familiar and frequently used conditions to be defined as *macros*. In Figure 4 (a), "BelowThreshold" and "AltitudeQualityOK" are both macros. The definition of the BelowThreshold macro is given in Figure 4 (b). A more refined (and more complex) version with several columns can be found in Figure 12.

4.2 Inter-component Communication

There should be a clear distinction between the inputs to a component, the outputs from a component, and the internal state of the component. Every data item entering and leaving a component is defined by the input and output variables.

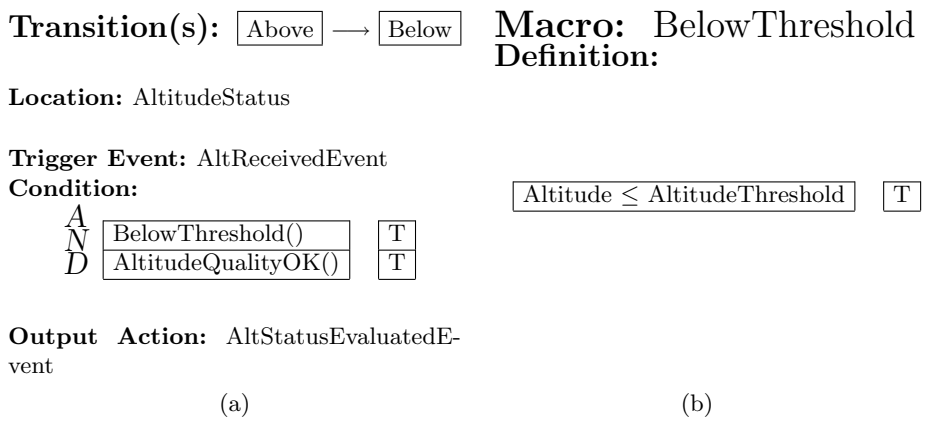

Fig. 4. A Transition and Macro from the ASW requirements

The state machine can use both input and output variables when defining the transitions between the states in the state machine. However, the input variables represent direct input to the component and can only be set when receiving the information from the environment. The output variables can be set by the state machine and presented to the environment through output interfaces.

RSML supports rigorous specification and analysis of system level inter-component communication. Communication in the framework occurs through simple messages consisting of a number of numeric or enumerated fields. Components in the system are connected via channels. Each component can have any number of incoming and/or outgoing channels. The formality of the specification allows us to automatically verify a specification for a number of simple safety and liveness constraints. For a more detailed description of the communication definitions and analysis procedures, the reader is referred to [13].

The NIMBUS environment is based on the ideas that (1) the engineers would like to have an executable specification of the system early in the project and (2) that as the specification is refined it is desirable to integrate it with more detailed models of the environment. Therefore, in the initial stages of the project, we want the executions to take their input from simple models, for example, text files or user input. As the specification is refined, the analyst can add more detailed models of the sensors and actuators, for example, additional RSML specifications or software simulations. In order to have a closed loop simulation, a model of the process can be added between the sensor and actuator models. Finally, when the specification has been refined to the point of defining the hardware interfaces, the analyst can execute it directly with the hardware. This hardware-in-the-loop simulation closes the gap between the prototype and the actual hardware. These ideas are illustrated in Figure 5.

5 Specification-based Prototyping with NIMBUS

A general view of an embedded control system can be seen in inside square of Figure 6. This model consists of the process, sensors, actuators, and the software controller. The process is the physical process we are attempting to control.

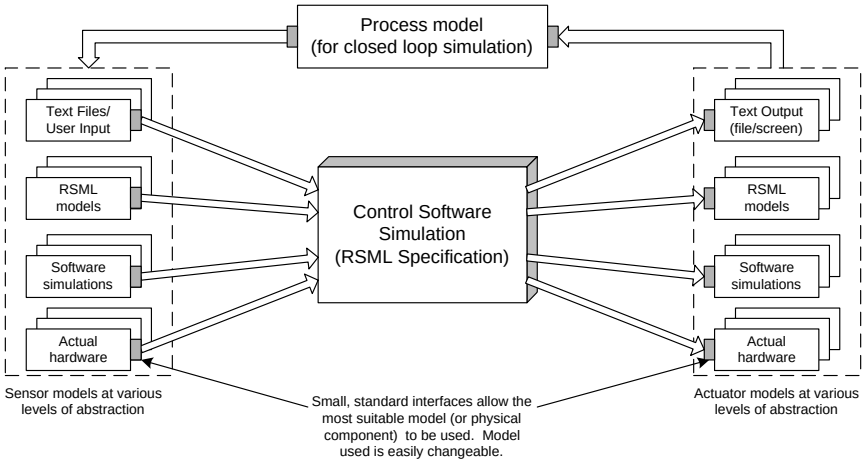

Fig. 5. The NIMBUS Environment

The sensors measure physical quantities in the process. These measurements are provided as input to the software controller. The controller makes decisions on what actions are needed and commands the actuators to manipulate the process. The goal of the software control is to maintain some properties in the physical process. Thus, understanding how the sensors, actuators, and process behave is essential for the development and evaluation of correct software. The importance of this systems view has been repeatedly pointed out in the literature [25, 19, 14].

To reason about this type of software controlled systems, David Parnas and Jan Madey defined what they call the four-variable model (outside square of Figure 6) [25]. In this model, the monitored variables (MON) are physical quantities we measure in the system and controlled variables (CON) are quantities we will control. The requirements on the control system are expressed as a mapping (REQ) from monitored to controlled variables. For instance, a requirement may be that *"when the aircraft drops below 2,000 ft, a device of interest shall be turned on."* Naturally, to implement the control software we must have sensors providing the software with measured values of the monitored variables (INPUT). The sensors transform MON to INPUT through the IN relation; thus, the IN relation defines the sensor functions. To adjust the controlled variables, the software generates output that activates various actuators that can manipulate the physical process; the actuator function OUT maps OUTPUT to CON. The behavior of the software controller is defined by the SOFT relation that maps INPUT to OUTPUT.

The requirements on the control system are expressed with the REQ relation; after all, we are interested in maintaining some relationship between the quantities in the physical world. To develop the control software, however, we are interested in the SOFT relation. Thus, we must somehow refine the system requirements (the REQ mapping) into the software specification (the SOFT mapping). The NIMBUS environment supports this refinement by allowing a progressively more detailed execution of the formal model throughout all stages of this refinement process.

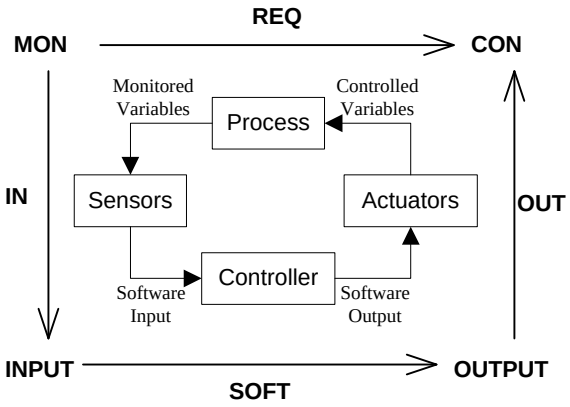

Fig. 6. The four variable model for process control systems

5.1 Structuring SOFT

The IN and OUT relations are determined by the sensors and actuators used in the system. For example, to measure the altitude we may use a radio altimeter providing the measured altitude as an integer value. Similarly, to turn on a device a certain code may have to be transmitted over a serial line. Armed with the REQ, IN, and OUT relations we can derive the SOFT relation. The question is, how shall we do this and how shall we structure the SOFT relation in a language such as RSML?

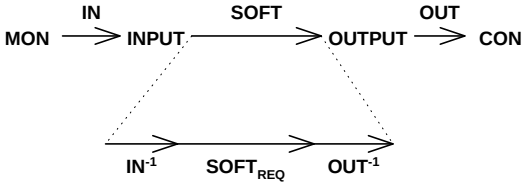

Fig. 7. The SOFT relation can be split into three composed relations. The SOFT$_{REQ}$ relation is based on the original requirements (REQ) relation.

The system requirements should always be expressed in terms of the physical process. These requirements will most likely change over the lifetime of the controller (or family of similar controllers). The sensors and actuators are likely to change independently of the requirements as new hardware becomes available or the software is used in subtly different operating environments; thus, the REQ, IN, and OUT relations are likely to change. If either one of the REQ, IN, or OUT relations change, the SOFT relation must be modified. To provide a smooth transition from system requirements (REQ) to software requirements (SOFT) and to isolate the impact of requirements, sensor, and actuator changes to a minimum, Steven Miller at Rockwell Collins has proposed to structure the software specification SOFT based heavily on the structure of the REQ relation [24].

Miller proposed to achieve this by splitting the SOFT relation into three pieces, IN^{-1}, OUT^{-1}, and SOFT$_{REQ}$(Figure 7). IN^{-1} takes the measured input

and reconstructs an estimate of the physical quantities in MON. The OUT^{-1} relation maps the internal representation of the controlled variables to the output needed for the actuators to manipulate the actual controlled variables. Given the IN^{-1} and OUT^{-1} relations, the SOFT_{REQ} relation will now be essentially isomorphic to the REQ relation and, thus, be robust in the face of likely changes to the IN and OUT relations (sensor and actuator changes). Such changes would only effect the IN^{-1} and OUT^{-1} portions of the software specification.

In the rest of this section we will illustrate how this framework for requirements specification and requirements refinement is used. We will also demonstrate how the NIMBUS environment supports dynamic evaluation of the various models throughout the refinement process. The result of this refinement process will be a formal specification of SOFT that, in NIMBUS, serves as a rapid prototype.

5.2 Explore the Requirements (REQ)

The first step in a requirements modeling project is to define the system boundaries and identify the monitored and controlled variables in the environment. In this paper we will not go into the details of how to scope the system requirements and identify the monitored and controlled variables. Guidelines to help identify monitored and controlled variables are covered in, for example, [24, 7, 17]

In the case of the altitude switch, we identified the aircraft altitude as one monitored variable and the commands that the ASW sends to the device of interest as a controlled variable. Both are clearly concepts in the physical world, and thus suitable candidates as monitored and controlled variables for the requirements model. The definition of the Altitude variable can be seen in Figure 2 and the graphical view of the requirements model is shown in Figure 3. In this paper, we will not discuss the details of the requirements model itself; instead, we focus on the refinement and execution of such models using NIMBUS.

The NIMBUS environment allows us to execute and simulate this model using input data representing the monitored variables and collect output representing the controlled variables. Input data could come from several sources. The simplest option for input is, of course, to have the user specify the values (either interactively, or by putting the values into a text file ahead of time). This scenario is illustrated in Figure 8(a). Unfortunately, it is often difficult to create appropriate input values since the physical characteristics of the environment enforce constraints and interrelationships over the monitored variables. Thus, to create a valid (i.e., physically realistic) input sequence, the analyst must have a model of the environment. Initially, this model may be an informal mental model of how the environment operates. As the evaluation process progresses, however, a more detailed model is most likely needed. Therefore, in this stage of the modeling we may develop a simulation of the physical environment. The NIMBUS architecture lets us easily replace the inputs read from text files with a software simulation emulating the environment. This refinement can be done without any modifications to the REQ model. For the ASW, we created a spreadsheet in Microsoft Excel to emulate the behavior of the aircraft (Figure 8(b)). This simple environmental model allows us to interactively modify the ascent and descent rates of the aircraft, and easily explore many possible scenarios.

5.3 Refine REQ to SOFT_{REQ}

From the start of the modeling effort, we know that we will not be able to directly access the monitored and controlled variables–we must use sensors and actuators.

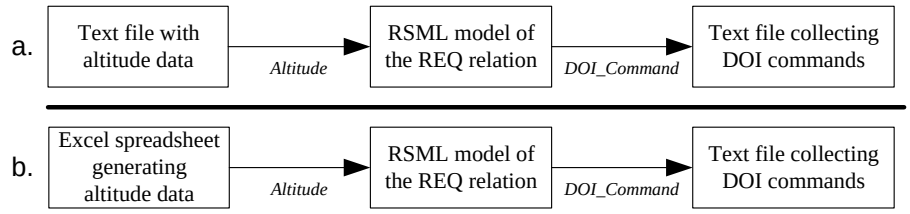

Fig. 8. The REQ relation can be evaluated using text files or user input (a) or interacting with a simulation of the environment (b).

Thus, when refining REQ to SOFT, we will not be able to use variables such as *Altitude*. At this early stage, we may not know exactly what hardware will be used for sensors and actuators; but, we do know that we must use something and we may as well prepare for it. By simply encapsulating the monitored and controlled variables we can get a model that is essentially isomorphic to the requirements model; the only difference is that this model is more suited for the refinement steps that will follow as the surrounding system is completed.

In our case, using a function, *MeasuredAltitude*(), instead of the monitored variable *Altitude* will shield the specification from possible changes in how the altitude measure is delivered to the software. By performing this encapsulation for all monitored and controlled variables we refine REQ to SOFT_{REQ}, a mapping from estimates of the monitored variables to an internal representation of the controlled variables.

5.4 IN, OUT, IN^{-1}, and OUT^{-1}

As the hardware components of the system are defined (either developed in house or procured), the IN and OUT relations can be rigorously specified. The IN and OUT models represent our assumptions about how the sensors and actuators operate. In the altitude switch we will use one analog and two digital altimeters. Thus, we will map the true altitude in the physical world to three software inputs (Figure 9).

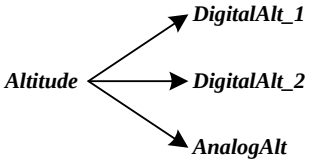

Fig. 9. The true altitude is mapped to three software inputs.

In the case of the digital altimeter, the altitude will be reported over an ARINC-429 low speed bus as a signed integer that represents a fraction of 8,192 ft. If we ignore inaccuracies introduced in the altitude measure and problems caused by the limited resolution of the ARINC-429 word, the transfer function for the digital altitude measures can be defined as

$$DigitalAlt = \frac{Altitude}{8192}$$

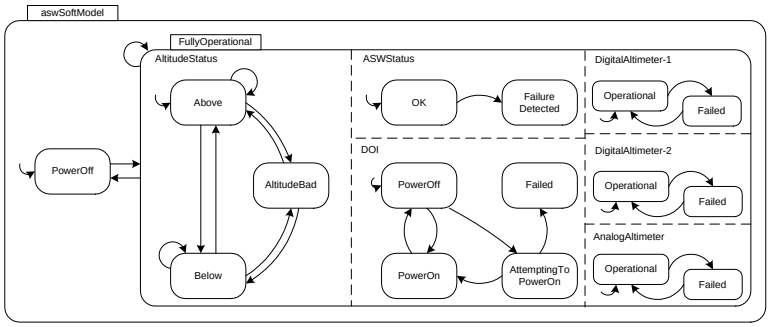

Fig. 10. The refined model of ASW with models of the three altimeters added.

The analog altimeter operates in a completely different way. Due to considerations of cost and simplicity of construction, the analog altimeter does not provide an actual altitude value, only a Boolean indication if the measured altitude is above or below a hardwired threshold (defined to be the same as the one required in the altitude switch). Assuming again an ideal measure of the true altitude, the transfer function for the analog altimeter could be modeled as

$$AnalogAlt = \begin{cases} Above \textbf{ if } Altitude > \text{Threshold} \\ Below \textbf{ if } Altitude \leq \text{Threshold} \end{cases}$$

In addition, all three altimeters provide an indication regarding the quality of the altitude measures.

Function MeasuredDigitalAlt(**Numeric** input) : **Numeric**
Equals input * DigitalAltMultiplier **If True ;**
Comment : DigitalAltMultiplier is defined as a constant 8192

Fig. 11. The function transforming the input from the digital altimeter to an estimate of the true altitude.

With the information about the sensor (IN) and actuator (OUT) relations, we can start refining the SOFT_{REQ} relation towards SOFT. In our case we must model, among other things, the three sources of altitude information and fuse them to one estimate whether we are above or below the threshold altitude. To achieve this, we refine the IN^{-1} relation in our model. The refined state machine can be seen in Figure 10. Internal models of the perceived state of the sensors have been included in the state machine. These new state machines are used to model IN^{-1}. Instead of the idealistic true altitude used when evaluating REQ, the specification now takes two digital altitude measures and one analog estimate of the altitude as input. The function estimating the true altitude from the digital altitude input is shown in Figure 11 and the modified *AboveThreshold*() macro is shown in Figure 12. Thanks to the structuring of the SOFT relation, this refinement could be done with minimal changes to the SOFT_{REQ} relation (compare the structure of the state machines in Figure 3 and Figure 10). As the components in the environment are developed, this process will be repeated for all inputs and outputs until a detailed definition of the SOFT relation is derived.

Macro: BelowThreshold
Definition:

			OR	
AND	MeasuredDigitalAlt(DigitalAlt_1) ≤ AltitudeThreshold	T	·	·
	DigitalAltimeter_1_OK()	T	·	·
	MeasuredDigitalAlt(DigitalAlt_2) ≤ AltitudeThreshold	·	T	·
	DigitalAltimeter_2_OK()	·	T	·
	AnalogAltitudeMeasure() = Below	·	·	T
	AnalogAltimeter_OK()	·	·	T

Fig. 12. Macro modified to handle the tree inputs instead of the true altitude as it did in the REQ model.

5.5 NIMBUS and Models of the Environment

When evaluating RSML specifications in NIMBUS, the analyst has great freedom in how he or she models the environment. When we evaluated the REQ model in Section 5.2, we used text files or a software simulation of the physical process to provide the RSML model with monitored variables and to evaluate the controlled variables. As the IN^{-1} and OUT^{-1} relations are added to the RSML model, the data provided (and consumed) by the model of the embedding environment must be refined to reflect the software inputs and outputs (INPUT and OUTPUT) instead of the monitored and controlled variables. This can be achieved in two ways; (1) refine the model of the physical process to produce INPUT and consume OUTPUT, or (2) add explicit models of the sensors and actuators to the simulation. In reality, the refinement of the environmental model and the SOFT relation progress in parallel and is an iterative process. The sensor and actuator models may be added one at a time and the interaction with different components may merit different refinement strategies. NIMBUS naturally allows any combination of the approaches mentioned above to be used.

In the case of the Altitude Switch, to simulate the refined SOFT relation (Figure 10) we modified our Excel model of the physical environment to produce digital and analog altitude measures (Figure 13(a)). The refinement was achieved by simply making Excel provide the three altitudes and applying the two transfer functions defined in Section 5.4 before the output was sent to the RSML model. Adding measurement errors to the sensor models can further refine the simulation of the ASW. For instance, by modifying the computation of the digital altimeter outputs to

$$DigitalAlt = \frac{Altitude}{8192} + \varepsilon$$

where ε is some normally distributed random error (easily modeled using standard functions in Excel), we can provide a more realistic simulation that includes the natural noise in the data from the altimeters.

As an alternative to refining the Excel model to include the altimeter models, we can explicitly add altimeter models to the simulation (Figure 13(b)). In our

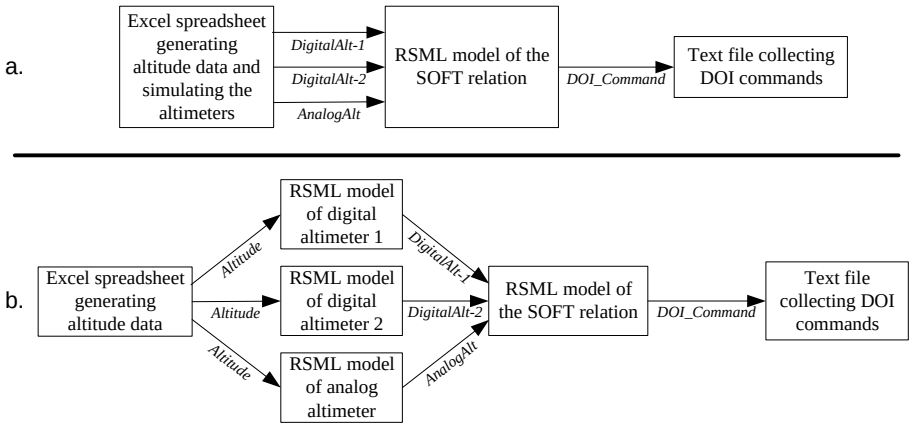

Fig. 13. Refined models of the environment; (a) using Excel to simulate the physical process as well as the sensors and (b) using Excel to simulate the physical process and RSML models to model the sensors.

case, we added altimeter models expressed in RSML. By adding explicit models of the sensors and actuators, we can easily explore how the software controller reacts to simulated sensor and actuator failures. Note that the integration of various different models with the RSML simulation of the control software does not require any modifications of the RSML model, the channel architecture of NIMBUS allows the analyst to easily interchange the component models comprising the environment.

As the refinement of the SOFT relation and the models of the environment progresses, we may at some point desire to perform hardware-in-the-loop simulation. Such simulations are easily accommodated in the NIMBUS framework. If we want to evaluate the ASW software requirements interacting with the actual hardware components, we can use any standard data acquisition card to access the hardware[1]. NIMBUS provides a collection of sample interfaces to the data acquisition card that can be easily modified to communicate with the desired hardware components. In the case of the ASW, we may want to take actual input from two digital altimeters, use a software simulation for the device of interest. (Figure 14)[2].

The use of hardware-in-the-loop simulation does not only provide a powerful evaluation of the proposed software system, we can also use NIMBUS to evaluate the physical system itself. For instance, by forcing the RSML model of the software requirements into unexpected and/or hazardous states, we can inject simulated software failures into the hardware system.

To summarize, NIMBUS provides a flexible framework in which a software requirements model expressed in RSML can be executed while it interacts with various models of the other components in a proposed system. NIMBUS supports the refinement of the REQ relation to a SOFT relation and allows easy interchange of components in the environment as the refinement takes place. It is important to recognize the difference between models which are good for representing the process versus models, like RSML models, which are good for mod-

[1] Currently, we are using the National Instruments DAQ 1200 series modules.

[2] Note that we have not yet had the opportunity to use actual digital altimeters in our simulations.

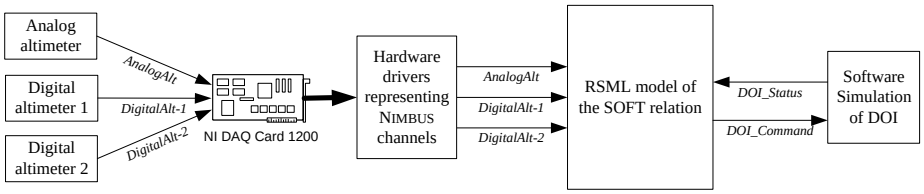

Fig. 14. An example of how hardware-in-the-loop simulation is achieved in the NIMBUS framework.

eling the software control of the process. Modeling the process itself accurately may require complex numerical functions, for example, generating normally distributed random errors. These types of functions are not and should not be within the scope of RSML. However, an accurate model of the process is key to the success of specification-based prototyping. This is the reason that NIMBUS provides the flexibility to integrate with many different models expressed in various ways, and one of the primary contributions of the NIMBUS environment.

6 Conclusion

Specification-based prototyping is an approach to requirements specification and evaluation that integrates the advantages associated with a readable and formal requirements specification with the power of rapid prototyping, while at the same time eliminates many of the current drawbacks with rapid prototyping.

To support our approach, we have developed the NIMBUS environment in which the requirements specification can be executed. In this flexible framework, software requirements models expressed in RSML can interact with either (1) user input or text files, (2) high-level RSML models of the components in the environment, (3) software simulations of the components (at varying levels of refinement), or (4) the actual physical components in the target system (hardware in the loop simulation). Since we support the execution of requirements models at various levels of refinement, we can evaluate the behavior of models ranging from high-level systems requirements to detailed requirements of the software. The flexibility of the NIMBUS environment allows us to seamlessly refine a system's requirements model to a software requirements model while continuously having the ability to execute and simulate the models in a realistic embedding environment. Furthermore, this flexibility allows the executable specification to serve at the prototype of the proposed system at each level of refinement. At the end of the refinement process we have a fully formal, analyzable, and executable model to use as a basis for production software development.

Our approach to requirements execution and system simulation has many advantages over previous approaches suggested for embedded systems because of the combination of three factors. First, RSML is a readable and easy to understand requirements modeling language. This simplicity allows the customers to be intimately involved in the specification and development of the requirements model, whereas currently they are often only involved in the evaluation of the executions and simulations based on a rapidly coded prototype developed in a standard programming language. Second, the capability to simulate the system as a whole enables early dynamic evaluation of system level properties such as safety, robustness, and fault tolerance. Third, the executable requirements specification is used as a high-level prototype of the proposed software. The dynamic behavior of the system can be evaluated through execution and simulation. Once

this behavior is deemed satisfactory, the resulting formal requirements specification is guaranteed to be consistent with the behavior of the prototype, and the requirements can be used as a basis for development of the production system. The guaranteed consistency between the prototype and the requirements specification eliminates the problems of inconsistent documentation commonly associated with prototyping [6].

We are currently investigating specification-based prototyping further. We are gathering experience from the use of NIMBUS and we are developing guidelines and a process for how to effectively take advantage of the opportunities presented with this type of environment.

References

1. L. Abraido-Fandino. An overview of REFINE 2.0. In *Proceedings of the second symposium on knowledge engineering, Madrid, Spain*, 1987.
2. J.M. Atlee and M.A. Buckley. A logic-model semantics for SCR software requirements. In S.J. Zeil, editor, *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'96)*, pages 280–292, January 1996.
3. B. Auernheimer and R. A. Kemmerer. RT-ASLAN: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 12(9), September 1986.
4. Valdis Berzins, Luqi, and Amiram Yehudai. Using transformations in specification-based prototyping. *IEEE Transactions on Software Engineering*, 19(5):436–452, May 1993.
5. W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J.D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.
6. A. M. Davis. Operational prototyping: A new development approach. *IEEE Software*, 6(5), September 1992.
7. S. Faulk, J. Brackett, P. Ward, and J Kirby, Jr. The CoRE method for real-time requirements. *IEEE Software*, 9(5):22–33, September 1992.
8. S. Gerhart, D. Craigen, and T. Ralston. Experience with formal methods in critical systems. *IEEE Software*, vol-11(1):21–39, January 1994.
9. S. Gerhart, D. Craigen, and T. Ralston. Formal methods reality check: Industrial usage. *IEEE Transactions on Software Engineering*, 21(2):90–98, February 1995.
10. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
11. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.
12. Mats P. E. Heimdahl and Nancy G. Leveson. Completeness and consistency in hierarchical state-base requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.
13. Mats P.E. Heimdahl, Jeffrey M. Thompson, and Barbara J. Czerny. Specification and analysis of intercomponent communication. *IEEE Computer*, pages 47–54, April 1998.
14. C. L. Heitmeyer, B. L. Labaw, and D. Kiskis. Consistency checking of SCR-style requirements specifications. In *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, March 1995.

15. C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.
16. K.L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering*, 6(1):2–13, Januaray 1980.
17. Michael Jackson. The world and the machine. In *Proceedings of the 1995 Internation Conference on Software Engineering*, pages 283–292, 1995.
18. B. Kramer, Luqi, and V. Berzins. Compositional semantics of a real-time prototyping language. *IEEE Transactions on Software Engineering*, 19(5):453–477, May 1993.
19. N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering*, 20(9):684–706, September 1994.
20. David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–354, April 1995.
21. David C. Luckham, James Vera, Doug Bryan, Larry Augustin, and Frank Belz. Partial orderings of event sets and their application to prototyping concurrent timed systems. *Journal of Systems Software*, 21(3):253–265, June 1993.
22. Luqi. Real-time constraints in a rapid prototyping language. *Computer Languages*, 18(2):77–103, 1993.
23. Luqi and V. Berzins. Execution of a high level real-time language. In *Proceedings of the Real-Time Systems Symposium*, 1988.
24. Steven P. Miller. Modeling software requirements for embedded systems. Technical report, Advanced Technology Center, Rockwell Collins, Inc., 1999. In Progress.
25. David L. Parnas and Jan Madey. Functional documentation for computer systems engineering (volume 2). Technical Report CRL 237, McMaster University, Hamilton, Ontario, September 1991.
26. P. Zave. An insider's evaluation of PAISLey. *IEEE Transactions on Software Engineering*, 17(3), March 1991.