SPECIFICATION BASED PROTOTYPING OF CONTROL SYSTEMS

Mats P.E. Heimdahl and Jeffrey M. Thompson
Department of Computer Science and Engineering
University of Minnesota, Minneapolis, MN
{heimdahl,thompson}@cs.umn.edu

Introduction

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 the system behavior, improves communication between the different parties involved in specification effort, allows empirical evaluation of alternative solutions, and is one of the more feasible ways of validating a system's behavior.

In this paper, we focus on an approach to simulation and debugging of formal software specifications for control systems called *specification-based prototyping* [1]. Within the context of specification execution and simulation, specification-based prototyping combines the advantages of traditional formal specifications (e.g., preciseness and analysis) with the advantages of rapid prototyping (e.g., risk management and early user involvement). The approach lets us refine a formal and executable model of the *system requirements specification* to a detailed model of the *software requirements specification*. 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 common problems associated with evolving the prototype into a production system are 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 an engineer to dynamically evaluate an RSML^{-e} (Requirements State Machine Language without events [1, 2, 3]) specification while interacting with (1) user input or text file input scripts, (2) RSML^{-e} models of the components in the embedding environment, (3) software simulations of the components, or (4) the physical components themselves (hardware-in-the-loop simulation).

Specification-based Prototyping

In *specification-based prototyping*, the specification starts as a high-level model of the *system requirements*. This model is then iteratively refined, adding more detail as the system becomes better understood. During each iteration, the specification is executable and can therefore be used as the prototype of the proposed system. Eventually, the system requirements will be well-defined and the system engineer must allocate requirements to particular hardware and software components within the system. At this point, the *system requirements* can be refined to the *software requirements* by adding descriptions pertaining to the actual hardware with which the software must interact.

Process-Control Systems

A general view of a process-control system can be seen in the inside square of Figure 1. This model consists of the process, sensors, actuators, and the software controller. The process is the physical process we are attempting to control. 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 [4, 5, 6].

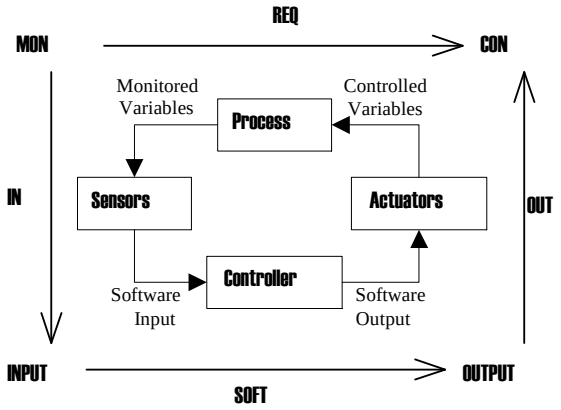

Figure 1: The four variable model for process control systems.

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 1) [1]. 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 requirements on 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 ultimately interested in maintaining some relationship between the quantities in the physical world and expressing the requirements in terms of the physical world makes them less sensitive to changes in the sensors and actuators. 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.

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?

The requirements as well as sensors and actuators are likely to change over the life of the system; thus, the REQ, IN, and OUT relations are all likely to change. If either one of the REQ, IN, or OUT relations change, the SOFT relation must be modified. We need to provide a smooth transition from system requirements (REQ) to software requirements (SOFT) and to isolate the impact of requirements, sensor, and actuator changes.

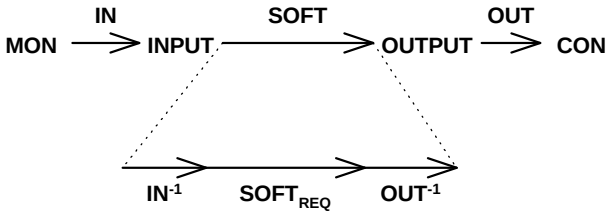

Figure 2: The SOFT relation can be split into three composed relations. The $SOFT_{REQ}$ relation is based on the original requirements (REQ) relation.

Steven Miller at Rockwell Collins has proposed to structure the software specification SOFT based heavily on the structure of the REQ relation in order to achieve these results [1, 4]. This structure involves splitting the SOFT relation into three pieces, IN^{-1}, OUT^{-1}, and $SOFT_{REQ}$ (Figure 2). 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 identical 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 specifi-

cation. Any requirements changes (changes to REQ) will now only affect $SOFT_{REQ}$.

Specification-based Prototyping in NIMBUS

The structuring techniques above can be applied to virtually any specification language (or even informal English requirements). Nevertheless, to realize the full potential of a prototyping-style development process, it is necessary to allow the analyst to fully evaluate the specification after each iteration. Therefore, a language which has a formal semantics and can be analyzed and executed during each iteration will have significant benefits over others that do not. Our analysis and execution capabilities are provided by an environment that we call NIMBUS [3].

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) as the specification is refined it is desirable to integrate it with more detailed models of the physical environment. Therefore, in the initial stages of the project, we want the executions to take their input from simple models, e.g., text files or user input. As the specification is refined, the analyst can add more detailed models of the sensors and actuators, e.g., additional RSML^{-e} specifications or software simulations. In order to have a closed loop simulation, a model of the physical 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 3.

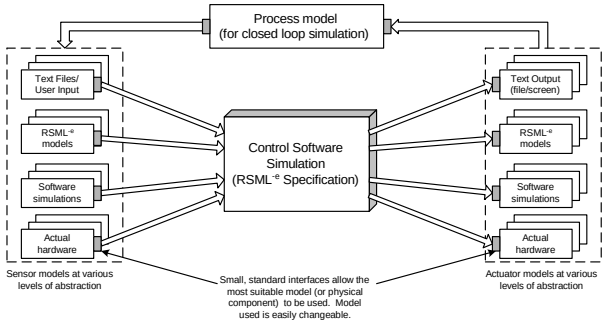

Figure 3: The NIMBUS Environment

The Altitude Switch

The Altitude Switch (ASW) is a (somewhat) 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.

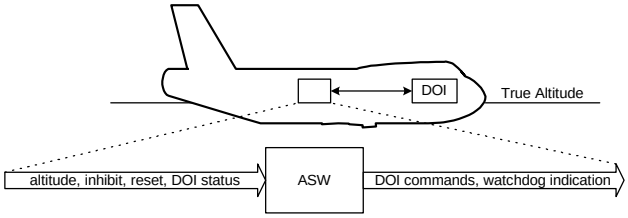

Figure 4: The ASW system in its environment.

The ASW and its environment are shown in Figure 4. 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.

The environment in which the ASW operates includes several features that make the system interesting. First, the ASW software does not have complete control over the DOI (Device of Interest). 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 analog and digital altimeters are significantly different in terms of the information that they provide.

The ASW Specification

In this section, we will illustrate the use of RSML^{-e}, NIMBUS and specification-based prototyping on our ASW case example. First, we describe how the ASW system requirements are formulated and how they can be evaluated in the NIMBUS environment. As we shall demonstrate, NIMBUS provides rich execution capabilities early in the requirements process. Next, we will demonstrate the refinement of the systems requirements to a software specification for the ASW and discuss the simulations and visualizations of that stage as well.

A Short RSML^{-e} Overview

An RSML^{-e} specification consists of a collection of *input variables*, *state variables*, *input interfaces*, *output interfaces*, *functions*, *macros*, and *constants*, some of which we will briefly discuss below.

In RSML^{-e}, the state of the model is the set of assignment histories of all *variables* and *interfaces*. The state information is used to compute the values of a set of *state variables*. These state variables can be organized in parallel or hierarchically to describe the current state of the system. Parallel state variables are used to represent the inherently parallel or concurrent concepts in the system being modeled. Hierarchical relationships allow *child* state variables to present an elaboration of a particular *parent* state value. Hierarchical state variables allow a specification designer to work at multiple levels of abstraction, and make models simpler to understand.

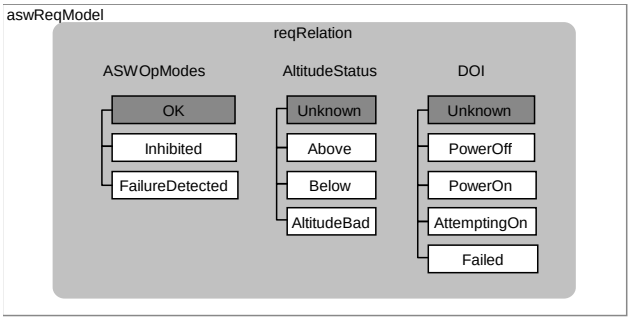

Figure 5: High-level ASW Model.

For example, consider the altitude switch. The state variable hierarchy used to model the requirements on this system could be represented as in Figure 5. This representation includes both parallel and hierarchical relationships of state variables. *AltitudeStatus*, *ASWStatus* and *DOI* are three parallel state variables, and all three are child state variables of *aswReqModel*.

Assignment relations in RSML^{-e} determine the value of state variables. These relations can be organized as *transitions* or *condition tables*. Condition tables describe under what condition a state variable *assumes* each of its possible values. Transitions describe the condition under which a state variable is to *change* value. A transition consists of a source value, a destination value, and a guarding condition. A transition is taken (causing a state variable to change value) when (1) the state variable value is equal to the source value, and (2) the guarding condition evaluates to true. The two relation types are logically equivalent; mechanized procedures exist to ensure that both functions are complete and consistent.

Input variables in the specification allow the analyst to record the values reported by the environment or various external sensors. They are assigned based on the messages received by input interfaces (discussed briefly below). The definition of the Altitude variable can be seen in Figure 6.

Input Variable

Altitude

Type: INTEGER

Units: ft

Initial Value: Undefined

Expected Min: 0

Expected Max: 40000

Figure 6: The definition of the Altitude input.

Interfaces encapsulate the boundaries between the RSML^{-e} model and the external world. 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 (state variables designated as outputs). 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 presented to the environment through output interfaces.

Macro

BelowThreshold

Parameters: NONE

Condition:

Altitude <= AltitudeThreshold	T

Figure 7: The macro in the REQ specification that determines if the altitude is below the threshold.

To further increase the readability of the specification, RSML^{-e} contains many other syntactic conventions. For example, expressions used in the predicates can be defined as functions and familiar and frequently used conditions can be defined as *macros*, e.g., BelowThreshold() (see Figure 7). *Functions* in RSML^{-e} are mathematical functions that are used to abstract complex calculations. A macro is simply a named condition that is used for frequently repeated conditions and is defined in a separate section of the document.

The System 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 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 graphical view of the requirements model is shown in Figure 5. We focus on the refinement and execution of such models using NIMBUS. The reader interested in the details of in RSML^{-e} and NIMBUS is referred to [2, 3, 4].

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). As the evaluation process progresses, however, a more detailed model is most likely needed. Therefore, 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 physical environment. This refinement can be done without any modifications to the REQ model. For the ASW, we created a simple spreadsheet in Microsoft Excel to emulate the behavior of the aircraft (Figure 8 (b)). The graphical interface for the Excel model is shown in Figure 9. This simple environmental model allows us to interactively modify the ascent and descent rates of the aircraft, and easily explore many behaviors of the ASW. Naturally, NIMBUS allows the user to record any interactive sessions as test cases for use at a later time.

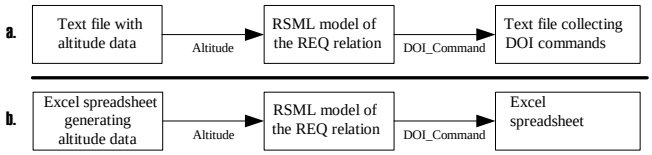

Figure 8: The REQ relation can be evaluated using text files or user input (a) or interacting with a simulation of the environment (b).

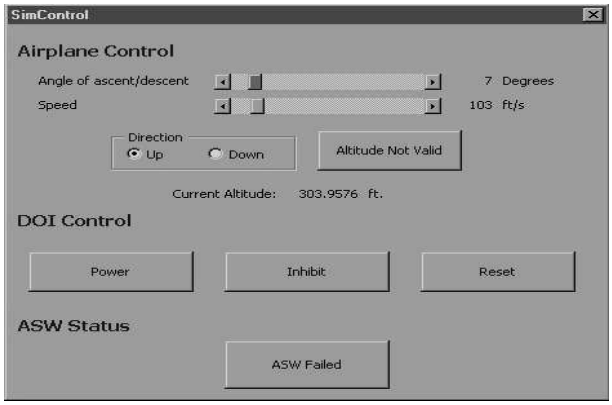

Figure 9: The ASW Excel system model used to evaluate REQ

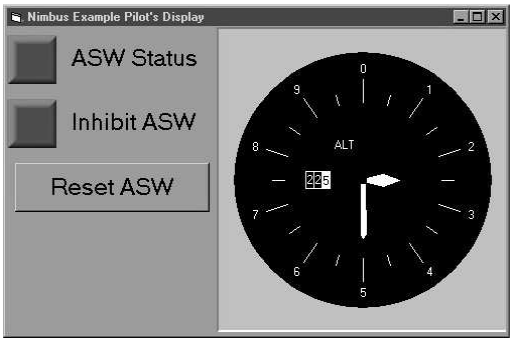

Figure 10: A mockup of the Pilot's display for the evaluation of the REQ model.

NIMBUS allows the user to visualize the system in many ways. The visualizations constructed using powerful user interface construction tools, for example, Visual Basic and can utilize the many third party ActiveX controls that are on the market. This makes it possible to construct rich visualizations, without expending large amounts of time or money. Thus, more development dollars can be used to ensure the quality of the specification. Figure 10 shows a mockup of a portion of the Pilot's Display that we quickly developed to illustrate this concept. Mockups like this, which are available while REQ is being developed, could be used to evaluate the po-

tential operator interface early in the development life cycle. This can allow the specifiers to catch many errors early and evaluate the REQ relation for, for example, the potential for mode confusion.

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. 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 identical to the requirements model.

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.

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 11).

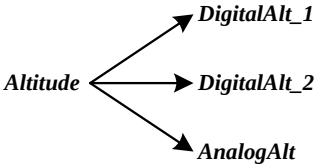

Figure 11: The true altitude is mapped to three software inputs.

In the case of the digital altimeter, the altitude will be reported over an ARINC-429 low speed bus as a signed floating-point value that represents the altitude as 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 $DigitalAlt = \frac{Altitude}{8192}$.

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

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

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

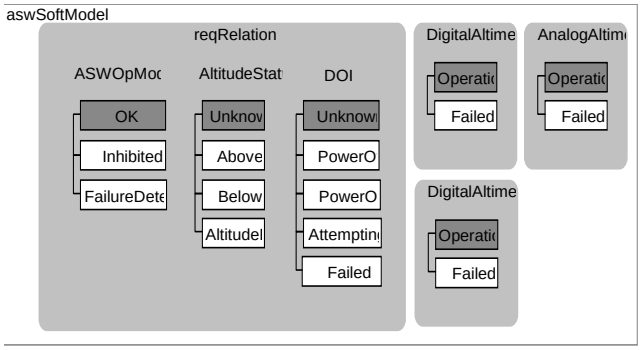

Figure 12: The refined model of ASW with models of the three altimeters added.

With the information about the sensor (IN) and actuator (OUT) relations, we can start extending our model 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 12. Internal models of the perceived state of the sensors have been included in the state machine. These 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 determination if we are below or above the threshold was encapsulated in a macro (see Figure 7) that can be modified to the macro *Below-*

Threshold() shown in Figure 13. 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 5 and Figure 12). 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

Parameters: NONE

Condition:

MeasuredDigitalAlt(DigitalAlt1) <= AltitudeThreshold	T	*	*
DigitalAltimeter1_Ok()	T	*	*
MeasuredDigitalAlt(DigitalAlt2) <= AltitudeThreshold	*	T	*
DigitalAltimeter2_Ok()	*	T	*
AnalogAltitudeMeasure() = Below	*	*	T
AnalogAltimeter_Ok()	*	*	T

Figure 13: Macro defined to handle the tree inputs instead of the true altitude as this macro did in the first version of the REQ model (compare to Figure 7).

Models of the Environment

When evaluating $RSML^{-e}$ specifications in NIMBUS, the analyst has great freedom in how he or she models the environment. When we evaluated the REQ model in an earlier section, we used text files or a software simulation of the physical process to provide the $RSML^{-e}$ model with monitored variables and to evaluate the controlled variables. As the IN^{-1} and OUT^{-1} relations are added to the $RSML^{-e}$ model, the data provided (and consumed) by the model of the environment must also 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 to be used.

In the case of the Altitude Switch, to simulate the SOFT relation (Figure 12) we modified our Excel model of the physical environment to produce digital and analog altitude measures (Figure 14 (a)). The refinement was achieved by simply making Excel provide the three altitudes and applying the sensor functions before the output was sent to the $RSML^{-e}$ 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.

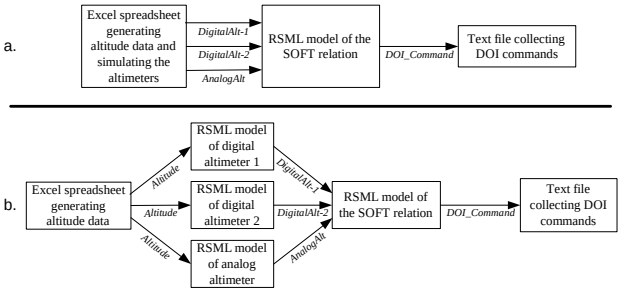

Figure 14: Refined models of the environment; (a) using Excel to simulate the physical process as well as the sensors and (b) using Excel to simulate the physical process and $RSML^{-e}$ models to model the sensors.

As an alternative to refining the Excel model to include the altimeter models, we can explicitly add altimeter models to the simulation (Figure 14 (b)). In our case, we added altimeter models expressed in $RSML^{-e}$. 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 sensor/actuator models with the $RSML^{-e}$ simulation of the control software does not require any modifications of the $RSML^{-e}$ model; the channel architecture of NIMBUS allows the analyst to easily interchange the component models comprising the environment [3].

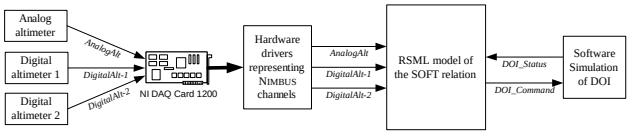

Figure 15: An example of how hardware-in-the-loop simulation is achieved in the NIMBUS framework.

As a final refinement, the analyst may wish to perform hardware-in-the-loop simulation. In the case of the ASW, we may want to take actual input from the digital altimeters and use a software simulation for the device of interest (Figure 15). Although we have not yet had the opportunity to use actual digital altimeters in our simulations, we have evaluated hardware-in-the-loop simulations in the mobile robotics domain [7].

Conclusion

Specification-based prototyping is an approach to requirements specification and evaluation that integrates the advantages of a readable and formal requirements specification with the power of rapid prototyping, while at the same time eliminating 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^{-e} can interact with either (1) user input or text files, (2) high-level RSML^{-e} 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.

Our approach to requirements execution and system simulation has many advantages over previous approaches suggested for process-control systems. First, RSML^{-e} is a readable and easy to understand requirements modeling language [2]. 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 [8].

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] Jeffrey M. Thompson, Mats P.E. Heimdahl, Steven P. Miller. Specification Based Prototyping for Embedded Systems. *Proceedings of the Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*. LNCS 1687; 163—179. September, 1999.

[2] Steven P. Miller. Modeling Software Requirements for Embedded Systems. Technical Report: Advanced Technology Center, Rockwell Collins, Inc.1999.

[3] Jeffrey M. Thompson. NIMBUS: *A Framework for Static Analysis and Simulation of System-level Inter-component Communication*. MS Thesis. Department of Computer Science and Engineering, University of Minnesota. December, 1999.

[4] D.L. Parnas and J. Madey. Functional Documentation for Computer Systems Engineering. *Science of Computer Programming*, vol-25, no-1; 41—61, 1991.

[5] N.G. Leveson, M.P.E. Heimdahl, H.Hildreth, and J.D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, vol-20, no-9,September 1994.

[6] C.L. Heitmeyer, R. Jeffords, and B.L. Labaw. Consistency checking of SCR-style requirements specifications. *ACM Transactions on Software Engineering and Methodology*, vol-5(3):231—261, July 1996.

[7] Jeffrey M. Thompson, Mats P.E. Heimdahl, Debra M. Erickson. Structuring Formal Control Systems Specifications for Reuse: Surviving Hardware Changes. *Proceedings of the Fifth NASA Langley Formal Methods Conference (Lfm2000). June, 2000.*

[8] Alan. M. Davis. Operational Prototyping: A New Development Approach. *IEEE Software*, vol-6, no-5. September 1992.