An Integrated Development Environment for Prototyping Safety Critical Systems*

Jeffrey M. Thompson and Mats P.E. Heimdahl
University of Minnesota
Department of Computer Science and Engineering
Minneapolis, MN 55455
{thompson,heimdahl}@cs.umn.edu

Abstract

The development of software for safety critical, embedded computer systems has been widely addressed in literature. Nevertheless, there does not currently exist any single environment which provides adequate support for all *of the following: static analysis, system simulation, animation and visualization, specification reuse, and refinement (from high-level requirements to implementation). In this paper, we present an overview of such an environment that is currently under development at the University of Minnesota concentrating on the prototyping capabilities and refinement model.*

1. Introduction

Prototyping languages, where successful, have typically occupied a niche. For example, Visual Basic has been highly successful in the development of user interface and database applications. When considering a possible prototyping method for safety critical systems the following criteria must be met:

- The language should support methods to assure the correctness of the system.
- A prototype of the system should be available early in the development life cycle.

In an embedded system, 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. These issues make validation and verification of software specifications for embedded systems particularly difficult.

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

In this paper, we focus on simulation and testing of the specification. Specifically, we describe how to integrate the advantages of readable formal specifications with the power of traditional rapid prototyping while avoiding many of the pitfalls of both. We call this approach *specification-based prototyping*.

The capability to dynamically analyze, or execute, the description of a software system early in a project has many advantages. Dynamic analysis helps stake holders 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.

This paper discusses the NIMBUS environment created at the University of Minnesota. NIMBUS, among other things, allows the analyst to start with a high-level specification of the system expressed in RSML (Requirements State Machine Language) and refine that model through the development cycle: adding models of the components in the environment, the process itself, even the actual hardware.

In the NIMBUS environment, we can use the formal, executable specification as the prototype. 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 (excluding real-time response) of the prototype and it is by definition updated as the prototypes evolve. Second, and perhaps more importantly, at any time the dynamic evaluation of the prototype can be augmented with formal analysis of the specification.

An overview of the NIMBUS environment concentrating on how it is used in prototyping is presented in this paper. Section 2 presents some background, as well as the goals of the NIMBUS environment. Next, in Section 3, we discuss related work. In Section 4, there is an overview of RSML and the NIMBUS environment itself. Section 5 discusses prototyping and model refinement with NIMBUS and the refinement model. Finally, Section 6 concludes.

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

2. Background

To fully specify, evaluate, and understand the software components of a system, essential information about the components interacting with the software must also be captured [6, 12, 13, 16]. Thus, a software requirements specification must include descriptions of the required software behavior as well as essential assumptions about the environment in which the software will operate.

Lutz found that a number of serious problems can result from incorrect environmental assumptions [21, 22]. Often, these problems involve misunderstandings about how the hardware operates, incompatibilities in the timing between the sending and receiving interfaces, failure to detect and respond to inputs outside the normal operating regime, and failure to prevent undesirable outputs from being generated [13, 21, 23]. These investigations indicated that to accurately evaluate a proposed software system it should be possible to execute a prototype of the software in the context of its embedding environment early in the development cycle.

In an ongoing project, we have developed a requirements engineering environment, called NIMBUS, that allows us to evaluate an RSML specification while interacting with (1) RSML models of the assumed environment, (2) software simulations of the environment, or (3) the physical environment itself. 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 an 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 environment.

We found that our initial RSML tool-set fell short of these goals. In addition, no currently available environment met the goals. Therefore, we have refined our specification language to better support the definition of inter-component communication and we have expanded our environment to support rapid prototyping of embedded systems.

3. Related work

Currently, early execution of a proposed software system is realized through either executable specifications or prototyping.

3.1. Executable specification languages

An executable specification language is a formally well defined, very high-level programming language. Languages such as PAISLey [24], ASLAN [3], and REFINE [1] are intended to replace requirements specifications, design specifications, and, in some instances, implementation code. Thus, most executable specification languages are intended to play many roles in the software development process. Executable specification languages have achieved some success and have been applied to industrial size projects. Many languages have elaborate tool-sets and support refinement of a high level specification into more detailed design descriptions or implementation code.

Nevertheless, current executable specifications 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 required of a notation if it is going to be usable as a requirements specification language. Several case studies have reported that current executable specification languages are difficult to use and are not suitable for requirements modeling. Furthermore, no currently available language provides support for high level specification of the interfaces governing the interaction between embedded software and the environment.

Notable exceptions to the languages discussed above are a collection of state-based notations. Languages such as Statecharts [7, 8], SCR (Software Cost Reduction) [11, 12], and the RSML [16], are very-high level and provide excellent support for inspections since they are relatively easy to use and understand for all stake holders involved in a specification effort. In addition, these languages allow automated verification of properties such as completeness and consistency [10, 11], and efforts are underway to model check state-based specifications of large software systems [2, 4].

3.2. Rapid prototyping

In software engineering, there are two main approaches to prototyping. One approach is to develop a draft implementation – a throw away or rapid prototype – in an attempt to learn more about the requirements on the software, 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 – evolutionary prototyping – 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. 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.

Notable examples of work in prototyping include PSDL [14, 20] and Rapide [19, 18]. 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 as much flexibility to integrate as easily with other tools as we desired.

In summary, no current approach provides satisfactory means for evaluating the behavior of embedded systems early during system development. Executable specification languages are not suitable as high-level requirements specification languages for embedded systems and they do not provide adequate support to specify system-level inter-component communication. Prototyping is applied too late in the life-cycle and often leads to unstructured and difficult to maintain systems.

4. RSML and the NIMBUS environment

RSML was developed as a requirements specification language specifically for embedded systems and is based on David Harel's Statecharts [7]. 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 [16].

In this paper, we present a very high level overview of RSML; thus, RSML may appear very similar to Statecharts. However, RSML differs from Statecharts in a number of critical ways, for example, states in RSML are akin to states in a finite state machine whereas states in Statecharts are typically used to control activities. A more detailed presentation of the semantics of RSML can be found in [16, 10].

4.1. Introduction to RSML

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

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 a simple railroad crossing could be represented as in Figure 1. This representation includes all three types of states. *Train_Crossing* is a parallel state with five direct children. All of these are compound states most which happen to contain only atomic states (*Up*, *Off*, etc.).

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

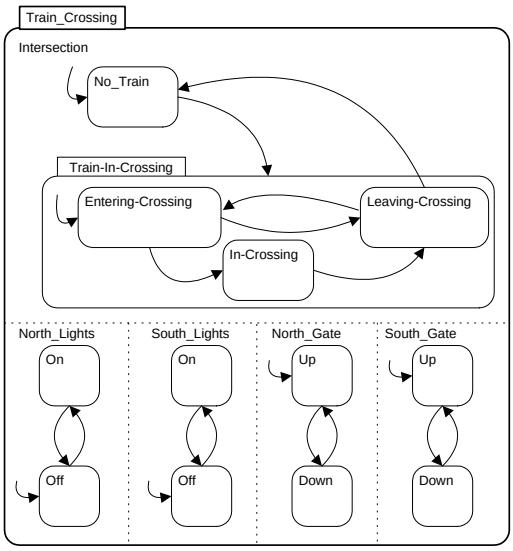

Figure 1. Train Crossing State Machine

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.

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).

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*. Also, guarding conditions are expressed in a tabular DNF (disjunctive normal form) notation called AND/OR tables.

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. 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 [9]. Com-

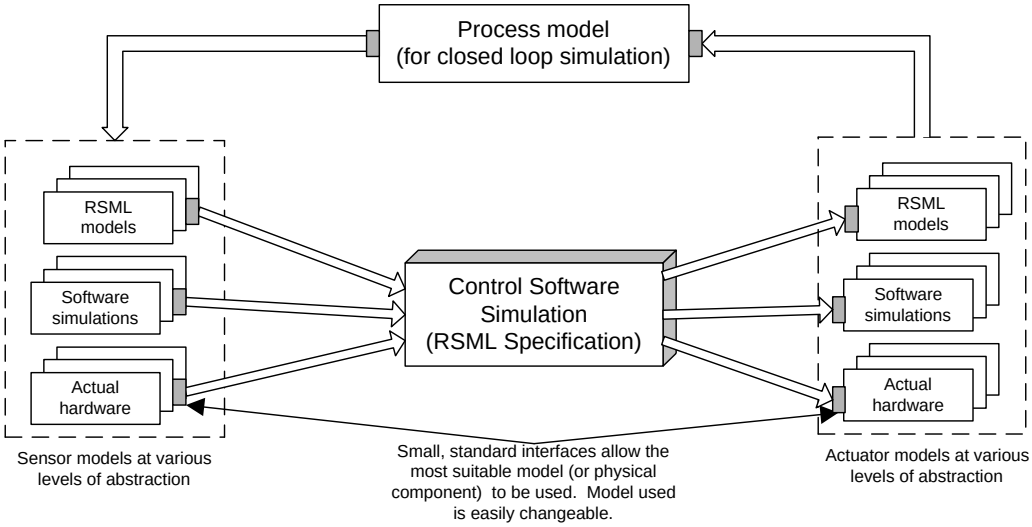

Figure 2. The NIMBUS Environment

munication in the framework occurs through simple messages consisting of a number of numeric or enumerated type fields. Components in the system are connected via channels. Each component can have any number of incoming and/or outgoing channels, each of which uses one of two possible communication mechanisms:

- **Send-Receive** communication where the message is transported across the channel with no buffering and causes an interrupt on the receiving side, and
- **Publish-Read** communication where the published message persists on the channel and is available at any time to the reader.

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 interested reader is referred to [9].

The constructs needed to rigorously define interfaces are an integral part of RSML. Thus, two RSML specifications can be connected and communicate with each other with no additional effort (assuming the interfaces between components are compatible). If an analyst wants to connect another tool, a software simulation, or possibly physical components to an RSML specification, we provide a small collection of classes one can use as a wrapper around the component that is to be integrated. Naturally, to achieve a full closed-loop simulation of a proposed system, a model of the physical process can be added as a separate component and used to connect the sensor and actuator models. These ideas are illustrated in Figure 2.

The NIMBUS environment rests upon a foundation build with COM (Component Object Model), Microsoft's standard for executable, reusable components. In order to add the capability to communicate in NIMBUS to any application that supports COM, all that is necessary is to register the NIMBUS dynamic link library on the system (a one time operation) and simply use the COM library to instantiate and initialize that application's end of the channel. In the RSML simulator, this process is automatically done given a valid RSML specification. For any other application, it can be done with only a few lines of code.

5. Prototyping in NIMBUS

Specification-based prototyping allows an iterative approach to building a formal specification. Initial versions of the specification can focus on the desired control behavior and ignore complicating details, such as sensor or actuator failures. The specification can be evaluated in an equally simplified environment containing failure free sensors and actuators. As the understanding of the system and its environment deepens, both the specification and the environmental models will be refined. The following paragraphs outline the activities that occur as the models are constructed and refined.

Prepare a high-level model. Initially, the objective of the analyst should be to clearly define the boundaries and high-level requirements of the proposed system. Thus, the model constructed should be in terms of environmental monitored and controlled variables. The focus is on the normal operating regime of the software controller and how to handle failures in the environment. This allows analysts and other stake-holders to get a clear idea of the primary function of the system, the primary exceptional conditions, and which system components interact with the software.

The interfaces defined in the specification serve to define the boundaries of the system and enumerate with which components the software will communicate. Initial definition of the interfaces are simplistic representations of the values that are to be input to the software controller. Some failure handling should be introduced into the system. The analysts and stake-holders should carefully consider the constraints of the system. Which failures in the

environment should the system handle? Which failure conditions are beyond the scope of the specification?

Interfaces for the external components are defined so that failure signals are an input to the RSML specification; that is, the RSML specification does not detect failures at this point, rather, it gets that information from a non-realistic, simplified version of the "environment." This allows the analyst to focus on how the software should respond to error conditions rather than how these conditions are detected. Input at this stage is most likely from text files or interactive user input.

Note that, even at this early stage of system specification, an executable prototype is available (provided by the RSML simulator). In this fashion, the analysts and stake-holders can actively evaluate and test the high-level requirements of the system early in the development cycle.

Model environmental assumptions. Specifications of each component in the environment are constructed, for example, models of the sensors and actuators. These models are simplified versions of each component, most likely written in RSML. The models focus on the normal operating modes of the components, but include outputs for error conditions as well.

RSML specifications of the components are useful when evaluating, for example, the controller's response to single and multiple component failures. The RSML component models can be forced into their various fail states to simulate failures in the environment. In addition, developing a detailed RSML model of the components will enrich the developer's understanding of the system. However, since RSML models are discrete and deterministic, they are not suitable for modeling some system components.

Refine the environmental models. Software simulations can offer a more accurate and/or desirable model of some components by allowing a more flexible expression of their functionality. For example, some system components are inherently continuous or require complex algorithmic descriptions not supported by RSML. Thus, representations of the components can be statistical models, numerical models, pre-recorded data from the actual system, etc. With the NIMBUS framework, integrating such software simulations to model the components in the environment involves little additional effort.

To fully evaluate the control behavior of a proposed system, the controller will be connected to detailed models of the sensors and actuators, and these models will be connected to a simulation of the process being controlled. Such an arrangement allows the analyst to perform longer-term evaluations of the control behavior without developing the actual code for the controller.

Evaluate the human-machine interface. Many control systems contain a human operator in some control or supervisory position. The interface between this human and the control software has a critical impact on not only the usability and acceptance of the system, but also on the safety of the system [15]. Many mishaps and accidents are attributed to operator error [17]. Even with a well-defined interface, if the operator's conceptual model of the control software and the system does not match the actual behavior of the software, a situation known as mode confusion may arise [17]. In this situation, the software is performing some control action while the operator believes the software is performing some other action. The result may be that the operator takes actions that nullify the software's control or, in the worst case, that bring the system into a hazardous state.

Even in very early stages of system development, the NIMBUS environment provides the means to easily evaluate different interface designs as well as evaluation for the potential of mode confusion. With NIMBUS, the interface can be viewed as merely another component of the system. It could be represented through a software simulation (for example, an interface mockup written with Visual Basic or some other interface builder) or even a hardware mockup of the interface.

Hardware in the loop simulation. Unfortunately, it can be the case that the models of the system components which were developed contain erroneous assumptions. Thus, connecting to a hardware system that obeys the physical laws governing the actual system is invaluable. For example, researchers in the field of robotics have found time and again that there is no substitute for experimentation with actual robots, as opposed to robot simulations. We believe that the same is true for safety critical systems. In addition, to evaluate the fault tolerance and error recovery capabilities of a system design, NIMBUS allows the analyst to easily inject simulated software malfunctions into the physical system.

Naturally, it may the case that using the actual system for prototyping is infeasible. For instance, the actual system may be unavailable or using it might be too dangerous. In that case, a scaled down version of the hardware system which nevertheless preserves the physics of the real hardware would provide similar benefits.

Amortize the investment. Naturally, there is quite a high cost associated with developing the sensor and actuator models as well as the model of the physical process. This, however, is a cost that can be amortized over many projects in the application domain. In addition, it is possible that detailed modeling of the components is *already* being done within the organization. Our architecture virtually assures that connecting an RSML simulation to a specification or simulation created with any other tool is quite simple. Therefore, the effective cost of developing the models is again spread across several projects.

These activities, done in concert, provide a powerful and flexible means of constructing embedded systems.

6. Conclusion

In this paper, we have presented an approach to requirements specification and evaluation that integrates the advantages associated with a readable formal requirements specification and the power of rapid prototyping, while at the same time eliminates many of the current drawbacks with rapid prototyping. The approach uses a state-based requirements specification language (RSML) to model the required

behavior of the software. We have developed an environment in which the requirements specification can be executed. In this flexible framework, software requirements expressed in RSML can interact with either (1) high-level RSML models of the components in the environment, (2) software simulations of the components (at varying levels of refinement), or (3) the actual physical components in the target system (hardware in the loop simulation).

Our approach to requirements execution and system simulation has many advantages over previous approaches suggested for embedded systems. First, RSML is a readable and easy to understand requirements specification language. This simplicity allows the customers to be intimately involved in the specification and development of the software, whereas currently they are often only involved in the evaluation of the executions and simulations. 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 [5].

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. Atlee and M. Buckley. A logic-model semantics for SCR software requirements. In S. 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] W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.

[5] A. M. Davis. Operational prototyping: A new development approach. *IEEE Software*, 6(5), September 1992.

[6] 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.

[7] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[8] 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.

[9] M. P. Heimdahl, J. M. Thompson, and B. J. Czerny. Specification and analysis of intercomponent communication. *IEEE Computer*, pages 47–54, April 1998.

[10] M. P. E. Heimdahl and N. G. Leveson. Completeness and consistency in hierarchical state-base requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.

[11] C. Heitmeyer, R. Jeffords, and B. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[12] K. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering*, 6(1):2–13, Januaray 1980.

[13] M. S. Jaffe, N. G. Leveson, M. P. Heimdahl, and B. E. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, 17(3):241–258, March 1991.

[14] 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.

[15] N. Leveson. *Safeware: System Safety and Computers*. Addison Wesley, 1995.

[16] N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering*, 20(9):684–706, September 1994.

[17] N. Leveson, J. Reese, S. Koga, L. Pinnel, and S. Sandys. Analyzing requirements specifications for mode confusion errors. In *Proceedings of the Workshop on Human Error and System Development*, 1997.

[18] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–354, April 1995.

[19] D. C. Luckham, J. Vera, D. Bryan, L. Augustin, and F. Belz. Partial orderings of event sets and their application to prototyping concurrent timed systems. *Journal of Systems Software*, 21(3):253–265, June 1993.

[20] Luqi. Real-time constraints in a rapid prototyping language. *Computer Languages*, 18(2):77–103, 1993.

[21] R. Lutz. An overview of REFINE 2.0. In *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1993.

[22] R. R. Lutz. Analyzing software requirements errors in safety-critical, embedded systems. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 35–46, January 1993.

[23] B. Melhart. *Specification and Analysis of the Requirements for Embedded Software with an External Interaction Model*. PhD thesis, University of California, Irvine, July 1990.

[24] P. Zave. An insider's evaluation of PAISLey. *IEEE Transactions on Software Engineering*, 17(3), March 1991.