Specifying and Analyzing System-Level Inter-Component Interfaces *

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

August 24, 2004

Abstract

In control systems, the interfaces between software and its embedding environment are a major source of costly errors. For example, Lutz reported that 20% - 35% of the safety related errors discovered during integration and system testing of two spacecraft were related to the interfaces between the software and the embedding hardware. Also, the software's operating environment is likely to change over time further complicating the issues related to system-level inter-component communication.

In this paper we discuss a formal approach to the specification and analysis of inter-component communication using a revised version of RSML (Requirements State Machine Language). The formalism allows rigorous specification of the physical aspects of the inter-component communication and forces encapsulation of communication related properties in well defined and easy to read interface specifications. This enables us to both analyze a system design to detect incompatibilities between connected components and use the interface specifications as safety kernels to enforce safety constraints.

Keywords: Inter-component communication, formal methods, requirements specification, static analysis, state-based specification, RSML, safety kernel.

*This work has been partially supported by NSF grants CCR-9624324 and CCR-9615088, University of Minnesota Grant in Aid of Research 1003-521-5965, and NASA grant NAG-1-2242.

1 Introduction

Writing and validating software requirements for control systems present particularly difficult problems. For example, the software is required to interact with a variety of analog and digital components in its environment, the software must be able to detect and recover from error conditions in the environment, and the software is often subject to rigorous safety and performance constraints.

The interfaces between the software and the environment to be controlled are a major source of costly errors. Lutz reported that 20% - 35% of the safety related errors discovered during integration and system testing of two spacecraft were related to these interfaces [24]. Problems often involve misunderstandings about how the hardware operates, incompatibilities in the timing between the sending and receiving side, failure to detect and respond to inputs outside the normal operating regime, and failure to prevent undesirable outputs from being generated [20, 24, 30]. In addition, the software's operating environment is likely to change over time, further complicating the issues related to system-level inter-component communication. Thus, it is imperative that a requirements specification for an embedded software system rigorously captures the interfaces and the communication between the software and its embedding environment. Furthermore, a specification language should support analysis techniques which help to assure that two components communicating over a channel have compatible communication definitions, that input and output value assumptions are compatible, and that the components satisfy communication related safety assertions.

In this paper we focus on our research with respect to inter-component communication, which is one aspect of an ongoing effort to provide an integrated approach to the development of critical systems. At an abstract level, an embedded control system can be viewed as a collection of physically distinct components communicating over unidirectional channels; This view was adopted in the RSML (Requirements State Machine Language) approach and used successfully to model TCAS II (Traffic alert and Collision Avoidance System II) [21]. TCAS II is a large commercial avionics system that is required by the FAA (US Federal Aviation Administration) on all aircraft that have more than ten seats. The components represent physically separate pieces of the system: a software controller, sensors and actuators (analog or digital), and physical processes. The connections between the components are defined through interface specifications that capture the essential information about the physical communication.

In the work described herein, we have extended and refined RSML to support rigorous specification and analysis of system-level inter-component communication. Preliminary results from this effort appeared in [13] and an abbreviated description of our view of inter-component communication appeared in [14]. This paper provides a detailed account of our effort and discusses the formal semantics of inter-component communication as we as detailed examples. The formality of the specification of how an RSML model interacts with its environment allows us to (1) execute and simulate a high-level specification of embedded control software under realistic environmental conditions and (2) automatically verify certain types of communication related constraints. The analysis procedures allow us to check a specification for the following properties.

1. Compatibility of the physical connection; are the timing assumptions between the sending and the receiving side compatible?
2. Compatibility of input and output variable values; are the value assumptions on the output from one component compatible with the input variables in another component?
3. Compliance with safety constraints; can an output be generated under certain conditions?

The formalization of the interfaces and communication, and the analysis capabilities enabled through that formalization are the focus of this paper. Note that our thinking is not confined to the RSML formalism, a similar approach could be incorporated in other formal modeling notations, for example, the SCR notation [18, 15].

Languages based on hierarchical finite state machines, for example, Statecharts, SCR (Software Cost Reduction), and the Requirements State Machine Language (RSML), have been successfully used to specify various types of embedded control systems [21, 34, 16, 18]. The languages are relatively easy to use, allow automated verification of properties such as completeness and consistency, and support execution and dynamic evaluation of the specifications. However, the support to rigorously specify and analyze the communication between physically distinct components in a system is not well developed. In addition, we want the inter-component communication mechanism to support execution and simulation of a specification while communicating with a realistic environment, for example, we want to be able to execute a specification and have it communicate with software simulations of sensors and actuators (or even communicate with the physical sensors and actuators themselves—hardware-in-the-loop-simulation). This type of execution and simulation is not well supported in any of the state-based approaches mentioned above.

One requirements engineering environment, called NIMBUS, developed at the University of Minnesota provides support for these execution activities [40, 41]. To enable adequate specification testing and simulation capabilities, the requirements must be executable while interacting with a realistic reproduction of the actual control system (or, if necessary, the actual control system itself). A key to success in this project is to provide a suitable way to specify the system-level inter-component communication. When starting the project we identified several properties the communication mechanisms of the language must posses.

Easy to understand and use: The basic features of the communication between the physical components in a system and the software components are often determined early in the project (that is, during the system design phase); thus, many different stake-holders will be involved in the design process, for instance, domain experts, the potential users of the system, and representatives from regulatory agencies. Therefore, the notation must be easily understood so that it will be used to capture the essential information about the inter-component communication.

Support capture of essential properties: Bonnie Melhart and Nancy Leveson [21, 30] have discussed a collection of fundamental assumptions about inter-component communication that should always be captured in the specification of an embedded system. Examples of particular interest for this paper include the following:

- Assumptions about the time-type of the communication (interrupt or continual) so that two components communicating are compatible.
- Assumptions about the expected minimum and maximum separation between messages on a channel so that timing violations can be detected and handled.
- Assumptions regarding the expected minimum and maximum value of input and output variables so that erroneous values can be captured and handled.

Our notation must support the capture of such assumptions. Furthermore, the notation must support analysis so that we can assure that two interfaces connected over a channel have mutually compatible assumptions.

Allow encapsulation: As a project progresses, changes to the software's embedding environment, and consequently its communication requirements, will inevitably occur. Thus, the communication specifications must allow us to encapsulate information that is likely to change. Also, by encapsulating information in well defined units we can parcel the formal analysis for certain properties [38, 47] and greatly reduce the complexity of the analysis for certain types of safety properties. We can view the interfaces as simple *safety kernels* [22, 35] and enforce various constraints in these kernels.

Enable realistic execution and simulation: Ideally, the developer of a control system should be able to test the controller in a realistic environment. While state-based languages are suitable for the specification of many parts of a control system system, we recognize that state-based techniques are not suitable to specify all system components. For example, state machines cannot accurately model sensors and actuators with continuous behavior. If high-fidelity models of continuous parts of a system are necessary, tools more suited for this type of modeling should be used.

The implementation of the interfaces that we have developed supports realistic system simulation. Each component in the system can be represented by an accurate model, be that an RSML specification, numerical simulation, statistical model, or whatever suits the the analyst. Using our environment, it is even possible to do hardware-in-the-loop simulation and have an RSML model interact with the sensors and actuators in the actual environment. We believe that this ability to treat system components as black boxes and concentrate on the interfaces in a accurate system simulation is key to providing a suitable environment for the development of process control systems.

Other approaches to high-level specification of control system behavior, in particular SCR used in the CoRE method [4, 15], advocate rigorous capture of information about the monitored and controlled variables in the software's environment. This work, with its roots in the pioneering specification of the control software for the A7 aircraft [19, 18], has helped shape our approach to the specification of inter-component communication. CoRE advocates a semi-formal capture of the interfaces between components and the assumptions about the various measured and controlled variables related to the interfaces. In more recent work [15, 17], Heitmeyer *et al.* provide a formal semantics of SCR. They view the monitored and controlled variables as variables shared between the model and the

environment. They do not have the capability to specify different types of communication (for example, interrupt versus polling) and cannot group variables together into messages—they assume that only one variable changes at the time. In our work, we take a more flexible and fully formal approach to the communication specification. The formalism allows us to execute specifications in a realistic environment as well as formally analyze certain aspects of the communication.

Recent work in software architecture has resulted in the definition of many architecture description languages (ADLs), for example, Darwin [25, 26], Rapide [23], SADL [32, 31], UniCon [36], and many others [6, 5, 39, 27, 28, 42, 3, 43, 7, 8, 1, 2]. These languages allow the software designer to specify the components in a software system, the type of interconnections between the components and the configuration of the components and connections. In [29], Medvidovic and Taylor discuss a framework for classification and comparison of architecture description languages. Medvidovic and Taylor underscore the importance, for ADLs, in treating the types of connections between components as first-class language entities (i.e., different connection types, for example, an SQL database, can be specified). However, in our case, we wanted to restrict the types of connections allowed between components to the ones relevant for requirements models of control systems. Furthermore, by their very nature, ADLs are a tool of the software designer and we wanted our interfaces to focus only on details which should be included at the requirements level. Finally, the software architecture community itself is new and there is "still little consensus in the research community on what an ADL is [and] what aspects of an architecture should be modeled by an ADL" [29]. For these reasons, we determined that using a pre-existing architecture description language would not be suitable to our purposes and that a new interface description based on our knowledge of process control systems would be more appropriate. In short, we have positioned our inter-component communication definitions between the overly simplistic view in SCR and the design oriented view in the architecture description community.

The *safety kernel* approach to enforcement of constraints is not unique to our work. Nancy Leveson *et al.* discussed the use of a safety kernel to enforce safety policies in safety critical systems [22]. The kernel centralizes the enforcement of safety policies and detection/recovery of safety violation in one small easily verifiable component. The notion of safety kernels responsible for policy enforcement has been further discussed by Kevin Wika and John Knight [45, 46]. These approaches, however, mainly address the design and implementation stages of development and do not discuss verification of safety properties in the high-level specification stage.

John Rushby has provided a detailed and formal discussion of the suitability of a kernel approach for safety enforcement [35]. He concluded that a kernel architecture is most suited to enforce negative properties, for example, that certain actions are not taken in some situations. A kernel approach is more limited when it comes to enforcing positive properties, for instance, that an action is always performed under certain conditions. The analysis work presented later in this paper uses the communication definitions as a simple kernel architecture and the analysis approach is largely inspired by Rushby's discussion; consequently, our work is mostly suitable to enforce negative properties.

The paper is organized as follows. The next section describes the RSML language, focusing on the definition of the interfaces and variables necessary for inter-component

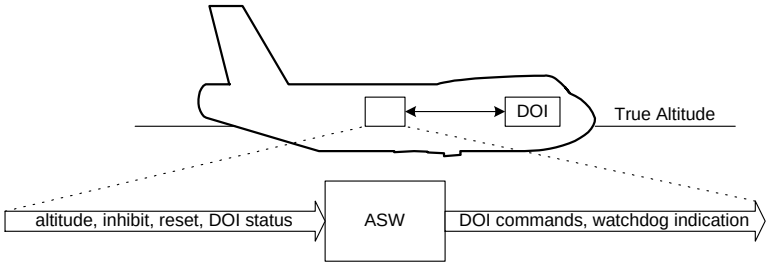


Figure 1: The ASW system in its environment

communication. In Section 3 we discuss how the communication definitions can be used to assure that the communication between components satisfies some properties inherent in inter-component communication. Sections 4 and 5 outline how our communication definitions can be viewed as simple kernels and how safety constraints can be expressed and verified. Finally, Section 6 provides a summary and conclusions.

2 Modeling Systems Using RSML

RSML was developed as a requirements specification language specifically for embedded control systems. 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.

The language is based on hierarchical finite state machines and is in many ways similar to David Harel's Statecharts [9, 10]. For example, RSML supports parallelism, hierarchies, and guarded transitions.

2.1 A Short Overview of RSML

An RSML specification consists of a collection of *states*, *transitions*, *variables*, *interfaces*, *functions*, *macros*, and *constants* which will be discussed in the remainder of this section.

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.

For example, consider the simple avionics system, called the altitude switch (ASW), where the system is responsible for turning on some device of interest (DOI), when the aircraft drops below a certain altitude threshold. The ASW and its environment are shown in Figure 1.

The state hierarchy modeling the high-level ASW requirements could be represented as in Figure 2. This representation includes all three types of states. *FullyOperational* is

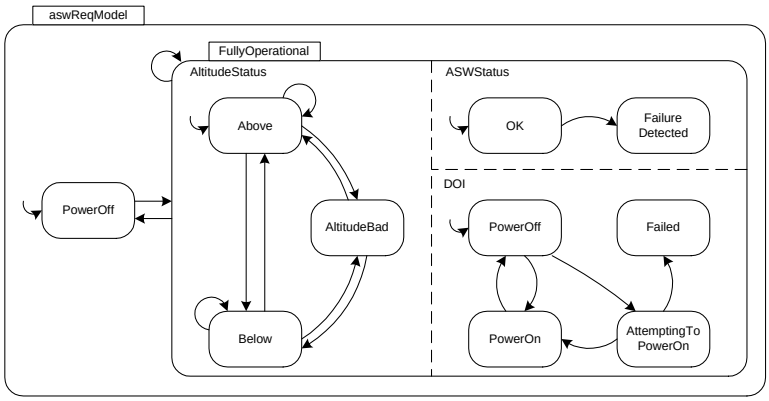


Figure 2: High-level ASW Model

a parallel state with three direct children (*AltitudeStatus*, *ASWStatus*, and *DOI*). All of these are compound states which contain only atomic states (*Above*, *Below*, etc.).

Transitions in RSML control the way in which the state machine can move from one state to another. The general form of any transition in RSML is show in Figure 3. As the figure shows, a transition consists of a source state, a destination state, a trigger event, a guarding condition, and a set of 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 set of events **E** will be produced.

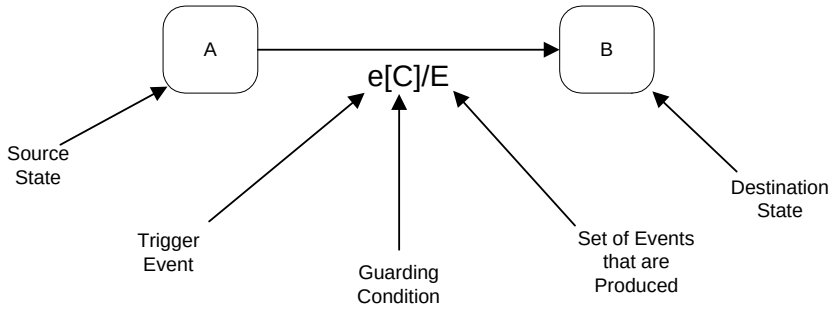
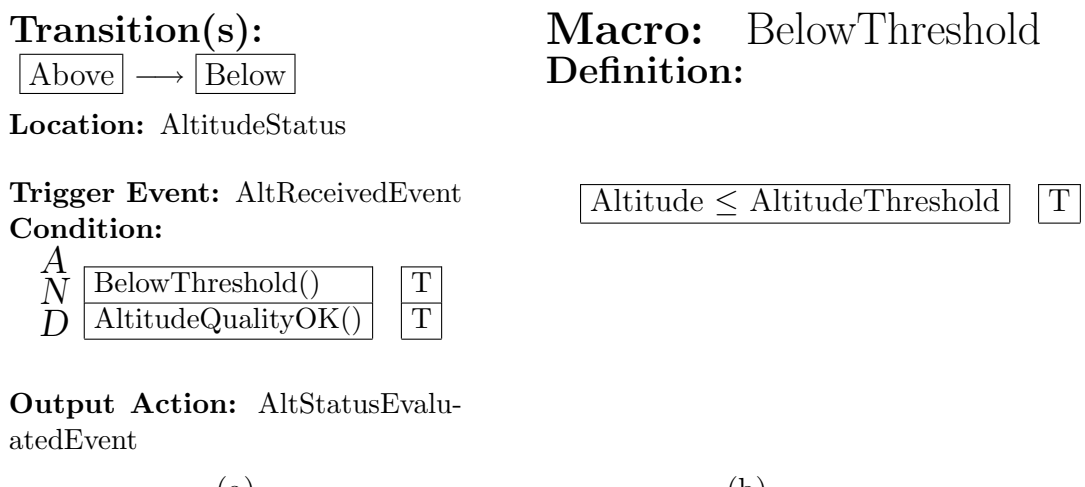


Figure 3: General RSML Transition

The guarding condition is simply a predicate logic statement over the various states and variables in the specification; however, during the TCAS project, the team that developed the specification (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 propositional 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. Figure 4 (a) shows a transition from the ASW requirements. Figure 5 shows a more complex example from the TCAS II requirements.



(a) (b)

Figure 4: A Transition and Macro from the ASW requirements

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 are match the truth values of the associated columns. A dot denotes "don't care."

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 *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).

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).

2.2 Systems and Components

In RSML we view a *system* as a collection of *components* connected by communication *channels*. A graphical representation (RSML notation) of a collection of system components and communication channels can be seen in Figure 6. The components are connected to the channels through *interfaces* and can send *messages* over the channels. A message is a collection of *fields* holding the atomic pieces of information communicated between the components. The only information flow between the components is through the unidirectional channels.

We make 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 describing

Transition(s): ESL-4 $\longrightarrow$ ESL-2

Location: Own-Aircraft $\triangleright$ Effective-$SL_{s\text{-}30}$

Trigger Event: Auto-SL-Evaluated-$Event_{e\text{-}279}$

Condition:

				OR
AND	Auto-$SL_{s\text{-}30}$ **in state** ASL-2	T	T	·
	Auto-$SL_{s\text{-}30}$ **in one of** {ASL-2,ASL-3,ASL-4,ASL-5,ASL-6,ASL-7}	·	·	T
	Lowest-$Ground_{f\text{-}241}$ = 2	·	·	T
	Mode-Selector **= one of** {TA/RA,TA-Only,3,4,5,6,7}	T	·	T
	Mode-$Selector_{v\text{-}34}$ = TA-Only	·	T	·

Output Action: Effective-SL-Evaluated-$Event_{e\text{-}279}$

Figure 5: A transition definition from TCAS II with the guarding condition expressed as an AND/OR table.

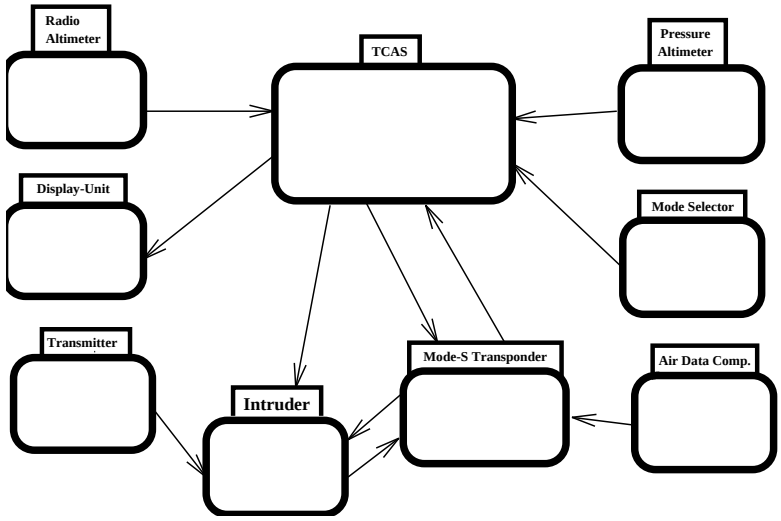


Figure 6: The components and channels in an avionics system.

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

Formally, the behavior of a finite-state machine can be defined using a next-state relation. In RSML, this relation is modeled by the transitions between states and the sequencing of events. For the purpose of this paper, we will focus our discussion on the message passing between physically separated components. We provide a definition of the global system state and how the various communication primitives defined in the next section effects that system state. For a rigorous treatment of the formal foundation of RSML and the details of the next state computation the reader is referred to [12]. A detailed description of the graphical notation and an account of the experiences from the TCAS II effort can be found in [21].

For the formal definitions in this paper we will use the Z notation [37]. The definitions in the following sections do not require extensive knowledge of Z; we will briefly describe any Z specific constructs we use.

The state of a components defined with RSML can be formally defined using the following types.

$$[Config, Message, Variable, Name]$$

The set *Config* is the set of all possible states the hierarchical state machine can be in. Because of the hierarchical and parallel structure of these types of machines there are many restrictions of which states are allowed. A discussion of these restrictions is, however, beyond the scope of this paper and has been extensively covered elsewhere [12]. *Message* is the set of messages that can be passed over the channels in the system, *Variable* is the set of allowable variable values associated with a component, and *Name* is the set of allowed identifiers.

Component

$$\begin{aligned}
&Configuration : Config \\
&InVar : Name \mapsto \text{seq } Variable \\
&OutVar : Name \mapsto \text{seq } Variable \\
&InMessage : Name \mapsto \text{seq } Message \\
&OutMessage : Name \mapsto \text{seq } Message \\
&InInterface : Name \mapsto InMessage \\
&OutInterface : Name \mapsto OutMessage
\end{aligned}$$

$$\begin{aligned}
&\text{dom } InVar \cap \text{dom } OutVar = \varnothing \\
&\text{dom } InMessage \cap \text{dom } OutMessage = \varnothing \\
&\text{dom } InInterface \cap \text{dom } OutInterface = \varnothing
\end{aligned}$$

The *Component* schema above defines the global state of an RSML model to consist of a configuration (the state of the hierarchical state machine) and a collection of sequences capturing the input and output histories of the component in question. *InVar* and *OutVar*

are mappings where a unique variable name is mapped to a sequence containing the trace of input or output variables received to or sent from the component. The relations *InMessage* and *OutMessage* pair unique message names with the message sequences in and out of the component. We make a distinction between variables and messages since a message can contain a collection of fields that will be assigned to different variables.

An interface in RSML is a connection point where a channel is connected to a component. Each interface accepts a message of a specific type. The *InInterface* and *OutInterface* relations match unique input and output interfaces to the message sequences that can be received or sent over the interfaces.

We define an initial state where all sequences are empty.

InitComponent

$$Component'$$

$$Component' = InitComponent$$
$$\forall x \in \operatorname{ran} InVar' \mid x = \langle\,\rangle$$
$$\forall x \in \operatorname{ran} OutVar' \mid x = \langle\,\rangle$$
$$\forall x \in \operatorname{ran} InMessage' \mid x = \langle\,\rangle$$
$$\forall x \in \operatorname{ran} OutMessage' \mid x = \langle\,\rangle$$

In the next section we describe the effect of message passing between two components as changes in the state of the sending and receiving side of a unidirectional channel.

3 The RSML Communication Model

In our formal definition of the RSML communication mechanisms we use a two tired approach. We will use Z to formally define the semantics of the RSML communication primitives. These primitives are then used to define the semantics of well structured communication interfaces suitable for a clear and easy to use description of system-level inter-component communication.

3.1 Communication Semantics

To introduce our approach to communication, we first provide an informal overview of the communication primitives. The formal definition appears later in this section.

All communication in a control system can be modeled as either interrupt driven or continual. Both types of communication are asynchronous with a non-blocking send; however, in the continual case one message is buffered (i.e., persistent) on the channel whereas with interrupt driven communication there is no buffering. With these two basic communication mechanisms we can implement any other communication scheme we may be interested in, for example, stimulus response.

We have defined two primitives to model interrupt driven communication; SEND($\langle channel \rangle$, $\langle message \rangle$) and RECEIVE($\langle channel \rangle$, $\langle message \rangle$). Informally, in terms of RSML's *event-action* semantics, SEND is an action that sends the message $\langle message \rangle$ over the channel $\langle channel \rangle$. RECEIVE is a trigger event that occurs when the message $\langle message \rangle$

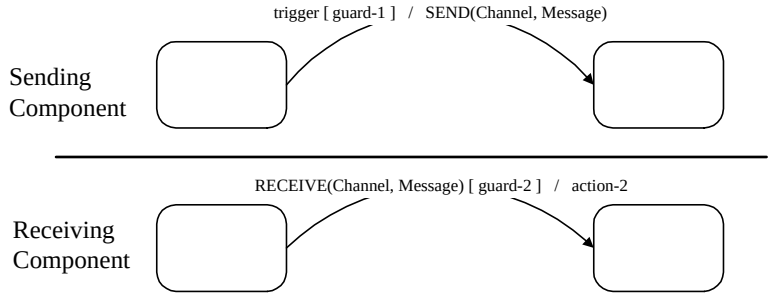


Figure 7: The SEND-RECEIVE pair as used in a state machine.

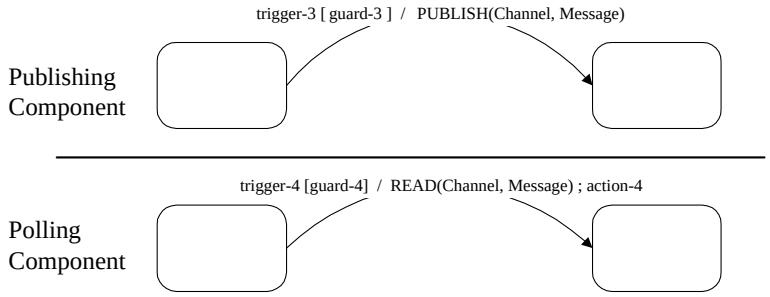


Figure 8: The PUBLISH-READ pair as used in a state machine.

is received over the channel ⟨*channel*⟩. The use of the SEND and RECEIVE primitives is illustrated in Figure 7. When the sending state machine takes the transition and issues the SEND command the RECEIVE event in the receiving component will be triggered. If the guarding condition (*guard-2*) is satisfied, the transition in the receiving component will be taken and the actions will be performed.

For continual communication we use two primitives; PUBLISH(⟨*channel*⟩, ⟨*message*⟩) and READ(⟨*channel*⟩, ⟨*message*⟩). Both primitives are actions in the RSML semantics. PUBLISH posts a message on the channel, and the information is persistent and remains available on the channel until it is overwritten by another PUBLISH command to the same channel. READ retrieves information from the channel. The information on the channel is unchanged by the read operation. Their use is illustrated in Figure 8.

Formal Semantics: A channel is a unidirectional one-to-one connection between an output interface of a component and an input interface of a (possibly the same) component. The definition can be seen in the schema below.

Channel

$$
\begin{array}{l}
Source : Component \\
SourceInterface : Name \\
Destination : Component \\
DestinationInterface : Name \\
\hline
SourceInterface \in \mathrm{dom}\ Source.OutInterface \\
DestinationInterface \in \mathrm{dom}\ Destination.InInterface
\end{array}
$$

The send action can be modeled as appending a message to the output message sequence associated with the appropriate output interface of the sending component. The send action also generates the message as an output to be used by the receive event that takes place in the receiving component. When a message is received it is simply added to the input message sequence of the appropriate interface[1].

Send

$$
\begin{array}{l}
\Delta Channel \\
MessageToSend? : Message \\
MessageSent! : Message \\
\hline
Source'.OutInterface(SourceInterface) = \\
\quad Source.OutInterface(SourceInterface) \frown \langle MessageToSend? \rangle \\
MessageSent! = MessageToSend?
\end{array}
$$

Receive

$$
\begin{array}{l}
\Delta Channel \\
MessageSent? : Message \\
\hline
Destination'.InInterface(DestinationInterface) = \\
\quad Destination.InInterface(DestinationInterface) \frown \langle MessageSent? \rangle
\end{array}
$$

A complete SEND-RECEIVE exchange is modeled as the SEND schema piping its output to the RECEIVE schema[2].

$$SendReceiveExchange \mathrel{\widehat=} Send \gg Receive$$

On the receiving side, as soon as a message is received it is processed by the state machine. Thus, we assume the synchrony hypothesis, that is, we assume that the state machine is infinitely faster than the environment and will always be able to complete the processing of one message before another message arrives. If we assume that the next-state

[1]The notation $\frown$ is the concatenate operator in Z. The expression *sequence-1* $\frown$ *sequence-2* concatenates the two sequences. The expression $\langle x \rangle$ creates a sequence containing x.

[2]The $\gg$ operator indicates piping of values. In the expression $X \gg Y$, any output variables (an identifier followed by !) in X is copied to the input variable (an identifier followed by ?) in Y with the same name.

computation of an RSML model (excluding the ability to receive messages) is defined by a Z schema named *RSMLBehavior*; then, the full behavior of an RSML model is defined by the schema composition below. This composition shows that as soon as a message is received, the next state computation is performed[3]. A description of the full formal semantics of RSML is beyond the scope of this paper, the interested reader is referred to [12] for an overview and [44] for complete coverage.

$$ReceiveMessage \mathrel{\widehat{=}} Receive \fatsemi RSMLBehavior$$

The PUBLISH and READ operations are straight forward to define. The publish action simply appends a new message to the output message sequence of the appropriate interface. This action is local to the publishing state machine. A read action in a state machine involves retrieving the last message appended to the output message sequence in the publishing state machine and append this message to the input message sequence in the reading machine. The read action always accesses the last message published on the channel. The two primitives are defined below.

Publish

$$\Delta Channel$$
$$MessageToPublish? : Message$$

$$Source'.OutInterface(SourceInterface) =$$
$$\quad Source.OutInterface(SourceInterface) \frown \langle MessageToPublish? \rangle$$

Read

$$\Delta Channel$$

$$Destination'.InInterface(DestinationInterface) =$$
$$\quad Destination.InInterface(DestinationInterface) \frown$$
$$\quad\quad last\langle Source.OutInterface(SourceInterface) \rangle$$

3.2 Interface Semantics

The communication primitives introduced in the previous section, together with the other constructs in RSML, are adequate to fully model a system component's required behavior as well as the system-level inter-component communication. Indiscriminate use of the communication primitives, however, may lead to unstructured and difficult to understand models. Thus, the communication with the environment should be encapsulated in well defined communication modules within each component. For example, in TCAS II all communication with one of the pilot's displays, called the RA-Display, should be confined to a small state machine dedicated to this task. By encapsulating the communication in dedicated state machines, the main parts of an RSML specification will be shielded from the inevitable changes in the embedding environment (Figure 9).

[3]The $\fatsemi$ operator indicates sequencing. $x \fatsemi y$ means we first apply x and then y.

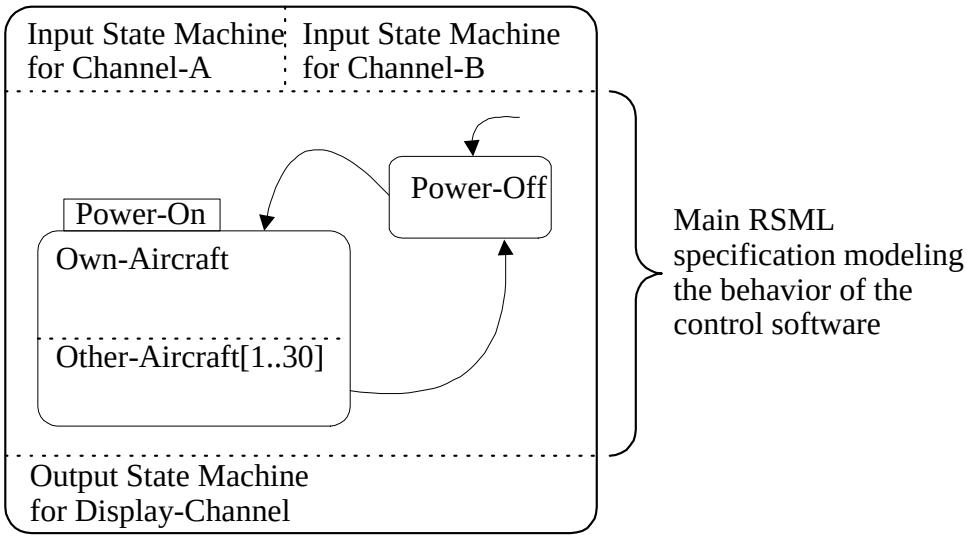


Figure 9: Dedicated communication state machines shield the RSML specification.

Our high-level RSML interface definitions are an abstraction of simple state machines using the basic RSML communication primitives. Naturally, the state machines could be directly used to specify the inter-component communication. But, since the state machines are very simple and only adds visual clutter to the graphical RSML model, we have chosen to use a purely textual notation to abstract away the state machines defining the communication. Also, the textual notation forces encapsulation of all communication information in the interface definitions. Leveson *et al.* successfully used a similar technique when specifying the communication mechanisms for TCAS II [21]. The definitions in this paper are an extension and refinement of their approach.

The interface definitions in Figure 10 are small examples of interface definitions in RSML. A realistic example taken from TCAS II is presented in Section 5. Interface definitions consist of two parts, (1) a physical interface definition that captures properties related to the physical aspects of the communication, for example, the channel name and timing assumptions, and (2) a collection of handlers that determine under which conditions we can either provide or accept messages over the channel. The physical interface definition is used to assure that components connected together have compatible properties, for example, that the expected minimum separation between messages at the RECEIVE side is less than or equal to the expected minimum separation at the SEND side. (We explain our compatibility criteria further in the next section.)

The interfaces in Figure 10 represent two distinct physical components communicating over the channel named *Altimeter-1-Channel*. The output interface is interpreted as follows. When a state machine in the main part of the specification generates the interface's trigger event (in this case *Send-Altitude-Event*) and a handler's guarding condition is satisfied, the output action in the handler is performed. For an input interface, when a message is received and the guarding condition on one handler is satisfied, the assignments defined in

Output Interface: Altimeter-Out-Interface

Channel: Altimeter-1-Channel
Trigger: Send-Altitude-Event
Max Separation: 0.45 seconds
Min Separation: 0.35 seconds

Handler-1
Condition:
True
Assignment(s):
Alt := MeasuredAltitude
Alt-Stat := SelfCheck()
Action: Send (Alt, Alt-Stat)

Input Interface: Altimeter-In-Interface

Channel: Altimeter-1-Channel
Trigger: Receive (Alt, Alt-Stat)
Max Separation: 0.5 seconds
Min Separation: 0.3 seconds

Handler-1
Condition:
Alt-Stat = Valid
Assignment(s):
DigitalAltitude := Alt
DigitalAltStatus := Alt-Stat
Action: Altitude-Received-Event

Handler-2
Condition:
Alt-Stat = Not-Valid
Assignment(s):
DigitalAltStatus := Alt-Stat
Action: Altitude-Bad-Event

Figure 10: Interface Example 1

that handler will be performed and the handler's output action is generated.

Both interfaces have a *min separation* and *max separation*. The *min separation* is the minimum amount of time that is required between messages. Similarly, the *max separation* is the maximum amount of time that can occur between messages. If there is no maximum separation, for example, if an event may never be generated, this is captured as an UNDEFINED upper bound. Figure 11 shows how the interfaces in Figure 10 are defined with state machines.

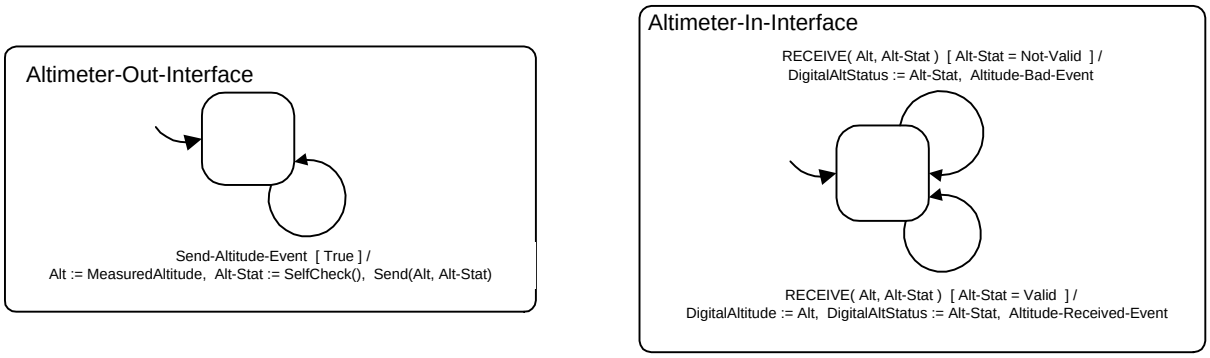


Figure 11: Definition of the interfaces in Figure 10 using RSML state machines and the basic communication primitives.

3.3 Working with Timing Assumptions

In Figure 11, the state machine does not capture nor enforce the timing properties defined in the physical definition of the interface. These timing properties merely capture *assumptions* about communication. To capture and manage violations of these assumptions, handlers

that address violations of the min separation and max separation should be added to the interface definition. To assist in expressing these timing constraints, we use *time expressions*, *time predicates* and *timeout events*. Time in RSML is represented as a tuple: (Hours, Minutes, Seconds, Milliseconds).

- Time expressions are simple expressions which result in a time. RSML allows the specification of time literals, for example, "3 hours and 2 seconds", as well as named time constants. In addition, the language supports getting the time of many occurrences in the system, such as the last time the model entered a state, assigned a variable, or processed input over a particular channel. Figure 12 shows a sample[4] of the time expressions in the RSML language. The formal definition of the time expressions is beyond the scope of this paper and can be found in [44].

- Time predicates allow the analyst to compare the values of two time expressions. The only time predicates that are allowed are less than or equal to ($\leq$) and greater than or equal to ($\geq$), which reflects the view of timing presented by Matt Jaffe [20] that states that no two events ever happen at exactly the same time. Thus, an equality operator is superfluous.

- Timeout events allows the analyst to schedule events to occur a specific time. Timeout events are of the form TIMEOUT(*start*, *duration*) where *start* is a time expression indicating the start time from which to start measuring the duration and *duration* is a time expression that indicates the length of time from the start time to when the timeout event would trigger.

Time Expression	**Meaning**
+,−	Normal arithmetic operations.
time_constant_name	The value of the named time constant.
time_literal	The value of the time literal.
TIME	The current simulation time (starting at 0:0:0:0 at the start of the simulation.
TIME(*event_name*)	The last time that the event occurred.
TIME(PREV(i) *event_name*)	The i^{th} previous time that the event occurred.
TIME(l ENTERED s)	The time that the state l entered its child s.
TIME(LASTIO)	The time of the last input or output on a particular interface.
TIME(MAXSEP)	The expected maximum separation between messages on a channel.
TIME(MINSEP)	The expected minimum separation between messages on a channel.

Figure 12: Representative sample of the time expressions allowable in RSML.

[4]Although there is the only one PREV expression in the table, it is possible to take PREV on the other occurrences, for example, TIME(PREV(i) LASTIO).

Figure 13 shows how the Altimeter-In-Interface in Figure 10 has been modified to capture violations of the min separation and max separation assumptions. Figure 14 shows the modified state machine capturing the meaning of this interface.

Input Interface: Altimeter-In-Interface

Channel: Altimeter-1-Channel
Trigger: Receive (Alt, Alt-Stat)
Max Separation: 0.5 seconds
Min Separation: 0.3 seconds

Handler-1

Condition:

Alt-Stat = Valid	T
TIME - TIME(LastIO) >= MINSEP	T
TIME - TIME(LastIO) <= MAXSEP	T

Assignment(s):
DigitalAltitude := Alt
DigitalAltStatus := Alt-Stat
Action: Altitude-Received-Event

Handler-2

Condition:
Alt-Stat = Not-Valid
Assignment(s):
DigitalAltStatus := Alt-Stat
Action: Altitude-Bad-Event

Handler-3

Condition:
TIME - TIME(LastIO) < MINSEP
Assignment(s): None
Action: Altitude-Time-Violation

Timeout-Handler-4: Timeout(LastIO, MAXSEP)
Condition: True
Assignment(s): None
Action: Altitude-Time-Violation

Figure 13: Modified Example

The high-level PUBLISH-READ communication is defined in much the same way as the SEND-RECEIVE. The only significant difference in the way that the handler transitions would look is that the PUBLISH and READ primitives would appear as actions on the transitions, not as the trigger as in the case of the RECEIVE primitive. Therefore, we will not discuss PUBLISH -READ communication further in this paper.

3.4 Input and Output Variable Definitions

Input and output variable definitions are similar to the interface definitions. Variables are typed and require the specification of assumed minimum and maximum values. An example

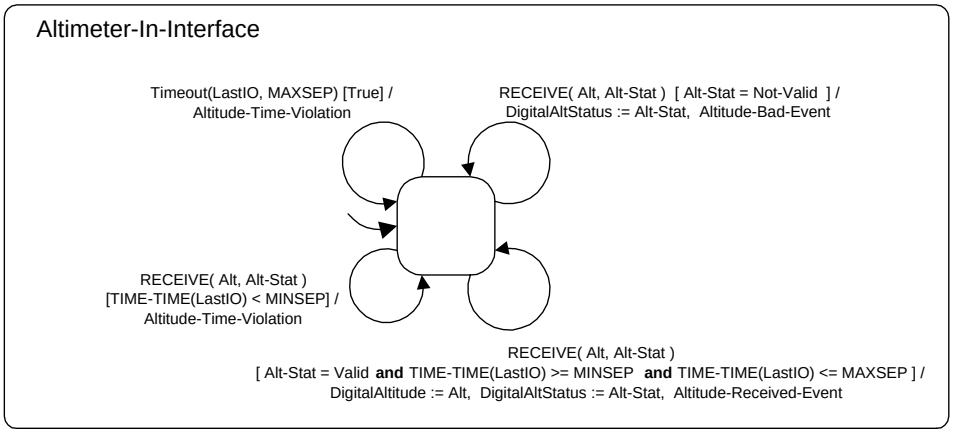


Figure 14: Definition of the interface in Figure 11 using an RSML state machine and the low-level primitives and timing properties.

of input and output variable specifications can be seen in Figure 15. Input variables can only be assigned by an input interface when a message is received from the environment. Output variables, on the other hand, can be assigned at any time. An output variable is assigned a new value when the state machine generates the trigger event for the variable (for example, in Figure 15, the Sample-Altitude-Event) and the variable is assigned the value of the *Value* expression (in this case, the function ReadAltitude). Both input and output variables can be assigned an optional *unit* indicating the physical entity the variable represents.

Output Variable: Measured-Altitude

Type: Integer
Expected Min: -1,000
Expected Max: 12,000
Trigger: Sample-Altitude-Event
Value: ReadAltitude()
Action: None
Unit: meters

Input Variable: DigitalAltitude

Type: Integer
Expected Min: -1,000
Expected Max: 12,000
Unit: meters

Figure 15: Variable example

Given the information about the physical communication, the assumptions on the input and output variables, and the handlers in the interfaces, we can perform various types of analysis.

4 Interface Compatibility Analysis

The interface definitions contain information about the physical aspects of the inter-component communication. To connect two interfaces over a channel, the interfaces must have compatible definitions as well as compatible input and output variables [20, 30]. Specifically,

- For each channel, the number of fields defined for the message in the source and destination specifications must be the same (types are dealt with below).
- A READ input handler must be connected to a PUBLISH output handler, and a RECEIVE input handler must be connected to a SEND output handler. That is, interrupt driven and continual communication cannot be mixed.
- For SEND -RECEIVE channels, the *max separation* of the input side must be greater than or equal to that of the output side. Also, the *min separation* on the input side must be less than or equal that of the output side.

In addition, each input handler in an input interface must match with *every* output handler in the corresponding output interfaces at the other end of the channel. This ensures that all possible combinations of output can be accepted by the input handler. For an input handler to match, it must assign the same number and type of input variables as the output handler sends on the channel. Each type of variable requires different checks as outlined below.

Integer variables

- The expected minimum of the output variable must be greater than or equal to the expected minimum of the corresponding input variable.
- The expected maximum of the output variable must be less than or equal to the expected maximum of the corresponding input variable.

If the output variable can take on larger or smaller values than the input variable, there is an incompatibility and our tool will report an error.

Enumerated variables

- If the types are the same, the variables are compatible. The types are the same if they have the name and same enumerations in the same order.
- If the types have the same name, but a different enumerations, a warning should be issued.
- If the types are not the same, and the type of the output variable has fewer possible values than the type of the input variable a warning will be issued stating that not all the enumerations of the input are used.

If the types are not the same, and the type of the output variable has more enumerations than the type of the input variable then "overflow" is possible and an error is reported.

Output Interface: Altimeter-Out-Interface

Channel: Altimeter-1-Channel
Trigger: Receive (Alt, Alt-Stat)
Max Separation: 0.35 seconds
Min Separation: 0.25 seconds
Trigger: Send-Altitude-Event

Handler-1
Condition:
True
Assignment(s):
Alt := MeasuredAltitude
Alt-Stat := SelfCheck();
Action: Send (Alt, Alt-Stat)

Output Variable: Measured-Altitude

Type: Integer
Expected Min: -3,000
Expected Max: 45,000
Trigger: Sample-Altitude-Event
Value: ReadMetricAltitude()
Action: None
Unit: ft

Figure 16: The output interface and the output variable in the new component are incompatible with the original input assumptions in Figures 10 and 15.

Consider the interface specifications and variables defined earlier. A change in the system requirements has prompted the replacement of the altimeter sending information over the Altimeter-Channel. To satisfy the new requirements, the new device is both faster (expected minimum separation 0.25 seconds) and is sending a variable with a wider range and different meaning than the previous device (from -3,000 ft to 45,000 ft). The output interface and the output variable associated with this new device are shown in Figure 16. Given this change in the assumptions about the inter-component communication, our tool will generate the error report in Figure 17. The new component is no longer compatible with the original input assumptions in Figures 10 and 15.

This analysis tool helps find problems in the initial systems architecture and highlights potential problems as the components in the system evolve over time. Forcing a consistent definition of the interfaces initially can help to prevent misunderstandings. This is especially useful on a large project which might have numerous components under simultaneous development.

5 Constraints and Constraint Verification

Since all communication is encapsulated in the interfaces, the guarding condition in a handler is effectively a precondition for the handler's communication to take place. This encapsulation acts as a simple *kernel architecture*; through these preconditions we can assure that no undesired outputs leave a component model and that no damaging inputs enter the model. The formality of the communication definition allows us to (1) assure

```
*********************************
 Report for channel Altimeter_Channel
*********************************

===> Error: Min separation of the output side (0.25 s) is less than
            the min separation of the input side (0.3 s).
===> Error: input variable DigitalAltitude has expected
            max less than the expected max of the output variable
            (Measured-Altitude) to which it is being assigned.
===> Error: input variable DigitalAltitude has expected
            min greater than the expected min of the output variable
            (Measured-Altitude) to which it is being assigned.
===> Warning: inconsistent units defined for variable DigitalAltitude.
            Output is (ft) and input is (meters).

*********************************
 End Report
*********************************
```

Figure 17: Error Report from the channel compatibility analysis tool.

that the input and output definitions are consistently and completely defined and (2) prove that communication related safety assertions hold in the model.

Output Interface: Display-Unit-Interface

Channel: Display-Channel
Trigger: Send-Traffic-Event[i]
Max Separation: 1.2 second
Min Separation: 0.8 second

Handler-1

Condition: For all j **in** {1..30}:

			OR
AND	i ≠ j	T	·
	Traffic-Display-Status[i] **in state** Waiting-To-Send	T	T
	Traffic-Display-Status[j] **in state** Waiting-To-Send	F	·
	Traffic-Score(Other-Aircraft[i]) ≥ Traffic-Score(Other-Aircraft[j])	·	T

Action: SEND(Advisory-Code[i])

Figure 18: Original definition of the communication with the pilot's display.

To illustrate our constraint verification we will use an example taken from TCAS II in Figure 18. The state machine model for TCAS II is required to model 30 intruding aircraft (modeled with the state machine Other-Aircraft in Figure 19). The model of each Other-Aircraft contains a state machine called Traffic-Display-Status. When TCAS has detected an intruder and has determined that the pilot needs to be notified, the state machine Traffic-Display-Status associated with that intruder will enter the state Waiting-To-Send.

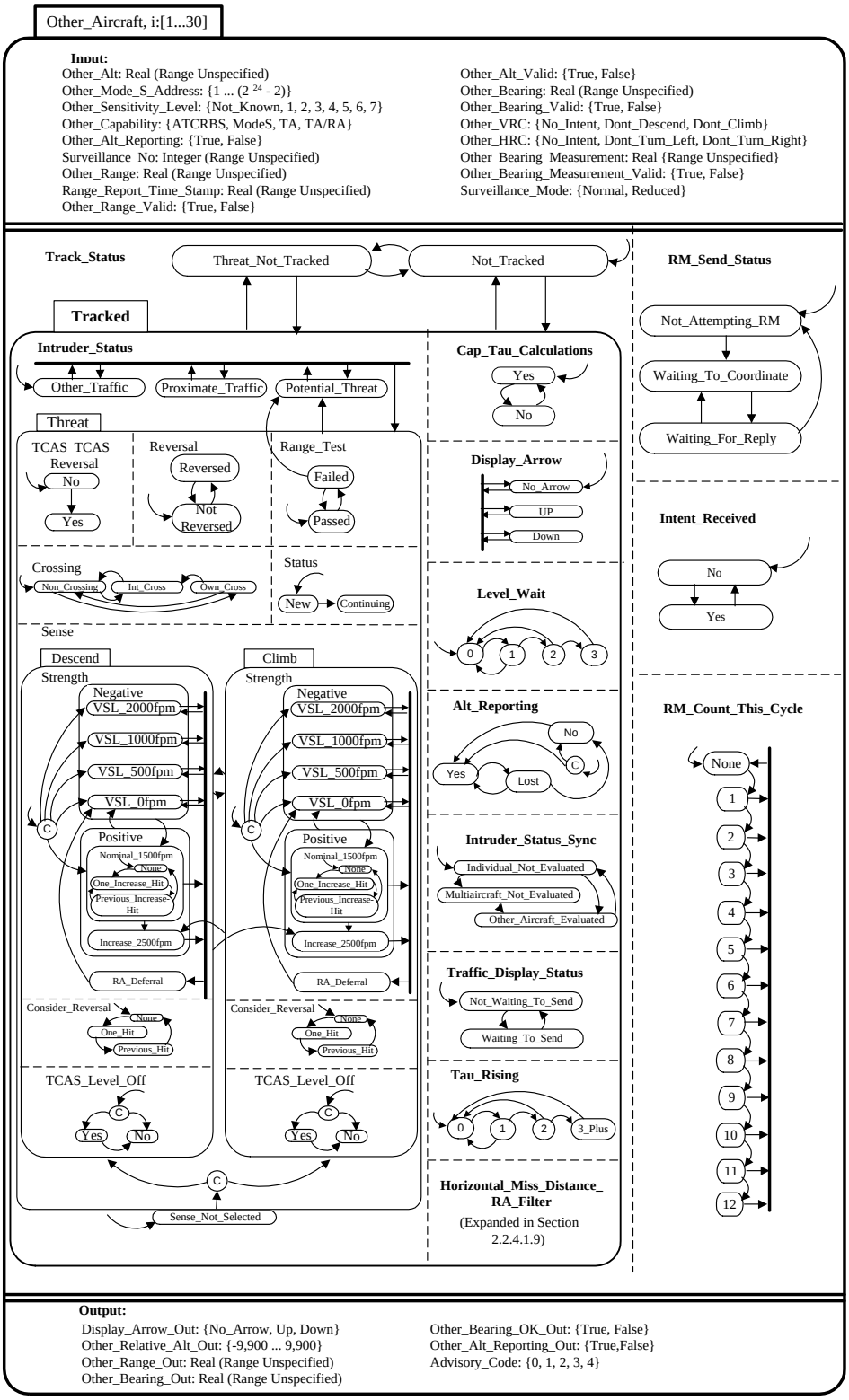


Figure 19: Model of an intruding aircraft

This indicates that TCAS is ready to send an advisory regarding this particular intruder to the pilot's display[5]. If TCAS tracks several intruders and needs to notify the pilot about more than one intruder (more than one Other-Aircraft model is in state Waiting-To-Send), the intruder model with the highest priority (Traffic-Score) takes precedence. The advisory relating to an intruder is contained in the variable Advisory-Code. The communication definition in Figure 18 defines the communication with the pilot's display. It is parameterized, any Other-Aircraft can generate a trigger event for this handler. The handler will simply be instantiated with the index of that intruder (the index is indicated with the i in the definition). Thus, Figure 18 tells us that Other-Aircraft[i] can only send an advisory to the pilot if there are no other aircraft ready to send (column 1) or there are no other aircraft with a higher traffic score (column 2).

The remainder of this section discusses how our communication formalism can be analyzed for completeness and consistency as well as various safety constraints.

5.1 Completeness and Consistency

In a previous investigation, we defined a collection of analysis procedures that assures that an RSML specification is complete and consistent [12]. The notion of completeness and consistency extends to the interface definitions. In this paper it suffices to state the completeness and consistency rules informally:

1. Within an interface definition, every pair of handlers triggered by the same event must have mutually exclusive guarding conditions; exactly one handler can be used at any time.
2. The logical OR of the guarding conditions on all handlers triggered by the same event within an interface definition must form a tautology; if an input arrives on a channel, it is always defined how this input will be handled.

Note, that the phrase "triggered by the same event" is significant in this context. This implies that, for example, in a receive interface which might have a number of regular handlers and several timeout handlers, the above conditions must hold for all the regular handlers (independent of the timeout handlers) and for each set of timeout handlers triggered by the same timeout event (independent of all the others).

Analysis procedures assuring that the criteria are satisfied are straightforward to automate: the guarding conditions on any two distinct handlers must be contradictory and the disjunction of the guarding conditions on the handlers triggered by the same event within each interface must form a tautology. Clearly, the interface in Figure 18 is incomplete since it only handles the case when we are actually allowed to send an advisory. On the other hand, the interface is by definition consistent since there is only one handler. A refined interface definition that is both complete and consistent can be seen in Figure 21. Our tool automatically generates the proof obligations for completeness and consistency. This, however, is not the focus of the paper and the interested reader must be referred to [11, 12] for a rigorous treatment of this topic.

[5]An advisory is a notification to the pilot; if the intruder is very close a resolution advisory will be displayed.

5.2 Safety Verification

In control systems there are often various safety constraints that must hold at all times. In TCAS, a safety constraint may be that we cannot remove a Resolution-Advisory from the pilot's display as long as the intruder that caused the advisory to be generated is declared to be a Threat (Other-Aircraft **in state** Threat)[6]. An intruding aircraft is declared to be a threat when a near mid air collision (NMAC) is considered imminent. We may, however, display a Resolution-Advisory against an intruder that is not a threat. An example of such a situation would be when a resolution advisory has only been displayed for a short time, the intruder flies past the own aircraft and is no longer considered a threat, but we want to keep the advisory for a few more seconds to provide a sense of continuity to the pilot.

Output Invariant:

The output: Advisory-Code[i]
can **only** be sent if
Condition:

		OR	
AND	Other-Aircraft[i] **in state** Threat	F	T
	Advisory-Code[i] = Resolution-Advisory	·	T

Figure 20: Safety constraint limiting when we can remove a resolution advisory from the pilot's display.

The constraint discussed above can be formally captured with an AND/OR-table as seen in Figure 20. Informally, the constraint in Figure 20 states that if we attempt to output an advisory for an intruder that is a threat it must be a resolution advisory. If all interactions with the environment are encapsulated in the interfaces, we will be able to verify constraints of this type by only considering the interface specifications.

The verification approach progresses in two simple steps. First, we determine which handlers can output the variable in which we are interested. Second, we show that the guarding condition (g) in those handlers implies the constraint (c), that is, that ($g \Rightarrow c$).

A similar approach can be used to prove simple negative constraints (an output is **never** sent under a certain condition). In this case, however, we want to show that all handlers that can send or publish that output have a guarding condition g that contradicts c ($\neg(c \wedge g)$).

We have augmented an existing execution environment and analysis tool for RSML with the capability to take communication related safety assertions as input. The tool generates proof obligations based on the rules outlined in this section. We generate proof obligations in the PVS (Prototype Verification System) [33] specification language and use the PVS theorem prover to perform the proofs [11]. The next section gives an example of the translation approach.

[6]Note that, although a reasonable constraint, this constraint was created for illustration only.

5.3 Generating Proof Obligations for PVS

PVS is a tool that provides an interactive environment for the development and analysis of formal specifications. It consists of a specification language, a parser, a type-checker, an interactive theorem prover, and various browsing tools.

Output Interface: Display-Unit-Interface

Channel: Display-Channel
Trigger: Send-Traffic-Event[i]
Max Separation: 1.2 second
Min Separation: 0.8 second

Handler-1

Condition: For all j **in** {1..30}:

				OR	
	i ≠ j	T	T	·	·
A	Traffic-Display-Status[i] **in state** Waiting-To-Send	T	T	T	T
N	Traffic-Display-Status[j] **in state** Waiting-To-Send	F	F	·	·
D	Traffic-Score(Other-Aircraft[i]) ≥ Traffic-Score(Other-Aircraft[j])	·	·	T	T
	Other-Aircraft[i] **in state** Threat	F	T	F	T
	Advisory-Code[i] = Resolution-Advisory	·	T	·	T

Action: SEND(Advisory-Code[i])

Handler-2

Condition: Exists at least one j **in** {1..30}:

						OR	
	i ≠ j	T	T	F	F	·	·
A	Traffic-Display-Status[i] **in state** Waiting-To-Send	F	F	F	F	T	T
N	Traffic-Display-Status[j] **in state** Waiting-To-Send	F	F	·	·	T	T
D	Traffic-Score(Other-Aircraft[i]) ≥ Traffic-Score(Other-Aircraft[j])	·	·	·	·	F	F
	Other-Aircraft[i] **in state** Threat	F	T	·	F	F	·
	Advisory-Code[i] = Resolution-Advisory	·	T	T	·	·	T

Action: None

Handler-3

Condition:

A	Other-Aircraft[i] **in state** Threat	T
N	Advisory-Code[i] = Resolution-Advisory	F
D		

Action: Assertion-Violation-Event

Figure 21: Modified definition of the communication with the TCAS display. This description is complete, consistent, and enforces the assertion in Figure 20

To illustrate our approach, consider the interface definition in Figure 18 and the assertion in Figure 20. Clearly, we cannot prove the assertion from the information provided in this interface specification. Furthermore, since the interface in Figure 18 only specifies when we are allowed to send but not when we are prohibited from sending, the interface

definition is incomplete. Figure 21 shows the same interface extended to handle the normal case when we are allowed to send an advisory, the case where we are not allowed to send an advisory, and the case where we have a safety violation. The rest of this section illustrates how we prove that this interface complies with the assertion.

```
%-----------------------------------------------------%
% Definition of the safety invariant 1 for the      %
% output variable Advisory-Code[i]                  %
%-----------------------------------------------------%

Output_Invariant: THEORY
BEGIN
IMPORTING TypeDefs
AdvisoryCode: VAR AdvisoryCodeType
OtherAircraft: VAR OtherAircraftType
i: VAR i_type
pred1?(OtherAircraft, i): bool = Threat?(OtherAircraft(i))
pred2?(AdvisoryCode, i): bool = ResolutionAdvisory?(AdvisoryCode(i))
OutputInvariant?(AdvisoryCode, OtherAircraft, i): bool =
      ( NOT pred1?(OtherAircraft, i)
         OR ( pred1?(OtherAircraft, i) & pred2?(AdvisoryCode, i) ))
END Output_Invariant
```

Figure 22: A PVS theory for the safety assertion in Figure 20.

```
%-----------------------------------------------------%
% Proof obligation for safety invariant 1           %
%-----------------------------------------------------%

OutputInterface: THEORY
  BEGIN

  IMPORTING TypeDefs, Handler1_OutputInterface,
            Output_Invariant

  AdvisoryCode: VAR AdvisoryCodeType
  TrafficDisplayStatus: VAR TrafficDisplayStatusType
  OtherAircraft: VAR OtherAircraftType
  TrafficScore: VAR TrafficScoreType
  i: VAR i_type

Handler1_implies_OutputInvariant: CONJECTURE
      Handler1?(AdvisoryCode, TrafficDisplayStatus,
              OtherAircraft, TrafficScore, i)
       IMPLIES
         OutputInvariant?(AdvisoryCode, OtherAircraft, i)

  END OutputInterface
```

Figure 23: A PVS theory the proof obligation for the constrint in figure 20.

Our tool generates a PVS theory for each handler and assertion in an RSML specification. We do this in a two stage process. First, we define each predicate in the AND/OR

table as a predicate in the PVS specification language[7]. Second, a predicate representing the full guarding condition (or assertion) is built from the individual predicates defined in the first stage. In PVS, the assertion in Figure 20 would be defined by the theory shown in Figure 22. The constants and type definitions used in the system are defined as separate theories and imported to the theory defining the assertion (or handler).

Since we are interested in proving that the variable Advisory-Code[i] with the value Resolution-Advisory can only be generated under certain circumstances, our analysis algorithm will identify Handler-1 in Figure 21 and generate its PVS theory.

The actual proof obligations are generated based on the criteria described in Section 5.2. Figure 23 shows the proof obligation for this example. In this case we want to show that the precondition (guarding condition) for the communication implies that the assertion is true (Figure 23). The translation to PVS is fully automated and the completion of proofs can in most cases be performed with a single predefined PVS strategy [11].

Since some interface specifications in our case study were parameterized, we chose PVS as our verification tool. PVS can, however, be unnecessarily powerful. In many cases methods such as formal inspections or a verification tool based on decision procedures will suffice. We are currently modifying our tools to allow the user to select the verification technique most suited for the problem at hand.

6 Summary and Conclusion

The correctness, safety, and robustness of the specification of a critical system are assessed through a combination of (1) rigorous specification capture and inspection, (2) formal analysis of the specification, and (3) execution and simulation of the specification. Any integrated approach to the specification of critical systems should support all three activities. In an ongoing project we are developing an environment with extensive support for these activities. In this paper we focused on one critical aspect of the specification process; the specification and analysis of the communication between the software and its environment.

We explicitly developed the approach to interface specification to support flexible execution and simulation of a specification in a realistic environment as well as allow us to formally verify certain properties of the communication. The communication mechanism, the inter-component interfaces, and the modeling language RSML all have a formal syntax and semantics. The rigorous specification of the interfaces and the communication allows us to analyze a system design and detect if there are incompatibilities between connected components, and the encapsulation of all communication of well defined interfaces enables us to view the interface specifications as simple safety kernels and enforce certain safety constraints in these kernels.

We have extended our tool set NIMBUS so that our notion of interfaces can be specified for other tools and applications. In this way, a model of an embedded control system need not consist of only RSML components but might include a variety of different types of simulations or even hardware. Allowing a heterogeneous simulation and execution environment will allow a flexible and realistic evaluation of a system and provide a powerful tool for dynamic evaluation of formal specifications.

[7] A predicate in PVS is a function with return type Boolean.

Acknowledgments

We would like to extend our thanks to Barbara C. Czerny of Delphi Automotive Systems for letting us use her RSML to PVS translator and for performing the PVS proofs. We would also like to thank the anonymous reviewers for their many helpful comments on earlier versions of this paper.

References

[1] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, May 1997.

[2] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[3] P. Binns, M. Engelhart, M. Jackson, and S. Vestal. Domain-specific software architectures for guidance, navigation, and control. *International Journal of Software Engineering and Knowledge Engineering*, 6(2), 1996.

[4] 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.

[5] D. Garlan, R. Allen, and J. Ockerbloom. Exploting style in architectural design environments. In *Proceedings SIGSOFT'94: Foundations on Software Engineering*, pages 175–188, December 1994.

[6] David Garlan. A introduction to the Aesop system, July 1995. http://www.cs.cmu.edu/afs/cs/project/able/www/aesop/html/aesop-overview.ps.

[7] M. Gorlick and A. Quilici. Visual programming in the large verses visual programming in the small. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 137–144, October 1994.

[8] M. Gorlick and R. Razouk. Using Weaves for software construction and analysis. In *Proceedings of the Thirteenth International Conference on Software Engineering (ICSE'91)*, pages 23–34, May 1991.

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

[10] 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.

[11] M. P.E. Heimdahl and B. C. Czerny. Using PVS to analyze hierarchical state-based requirements for completeness and consistency. In *Proceedings of the IEEE High Assurance Systems Engineering Workshop*, pages 252–262, 1996.

[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 and Jeffrey M. Thompson. Specification and analysis of system level inter-component communication. In *First International Conference on Formal Engineering Methods*, pages 192–201, November 1997.

[14] Mats P.E. Heimdahl, Jeffrey M. Thompson, and Barbara J. Czerny. Specification and analysis of intercomponent communication. *IEEE Computer*, pages 47–54, April 1998.

[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] C.L. Heitmeyer, J. Kirby, and B.G. Labaw. Applying the SCR requirements method to a weapons control panel: An experience report. In *Proceedings of the Second Workshop on Formal Methods in Software Practice (FMSP'98)*, Clearwater Beach, FL, March 1998.

[17] Constance Heitmeyer, James Kirby Jr., Bruce Labaw, Myla Archer, and Ramesh Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on Software Engineering*, 24(11):927–948, November 1998.

[18] 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.

[19] K.L. Heninger, J.W. Kallander, J.E. Shore, and D.L. Parnas. Software Requirements for the A-7e Aircraft. Technical Report 3876, Naval Research Laboratory, Washington, D.C., November 1978.

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

[21] 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.

[22] N.G. Leveson, T.J. Shimeall, J.L. Stolzy, and J.C. Thomas. Designing for safe software. In *Proceedings of the AIAA 21st Aerospace Sciences Meeting*, pages 1–5, 1983.

[23] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–733, September 1995.

[24] Robyn R. Lutz. Targeting safety related errors during software requirements analysis. In *SIGSOFT '93. First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 18(5) of *SIGSOFT Software Engineering Notes*, pages 99–106, December 1993.

[25] J. Magee, N. Dulay, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the Fifth European Software Engineering Conference (ESEC'95)*, pages 137–153, September 1995.

[26] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of the ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering*, pages 3–14, October 1996.

[27] N. Medvidovic, P. Oreizy, J.E. Robbins, and R.N. Taylor. Using object-oriented typing to support architectural design in the C2 style. In *Proceedings of the ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering*, pages 24–32, October 1996.

[28] N. Medvidovic, D.S. Rosenblum, and R.N. Taylor. A language and environment for architecture-based software development and evolution. In *Proceedings of the Twenty-first International Conference on Software Engineering (ICSE'99)*, pages 44–53, Los Angeles, CA, May 1999.

[29] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

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

[31] M. Moriconi, X. Qian, and R.A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, April 1995.

[32] M. Moriconi and R.A. Riemenschneider. Introduction to SADL 1.0: A language for specifying software architecture hierarchies. Technical Report SRI-CSL-97-01, Carnegie Mellon University, March 1997.

[33] S. Owre, N. Shankar, and J.M. Rushby. *The PVS Specification Language*. Computer Science Laboratory; SRI International, Menlo Park, CA 94025, beta relese edition, April 1993.

[34] R.Bharadwaj and C. Heitmeyer. Applying the SCR requirements specification method to practical systems: A case study. In *The Twenty-first Software Engineering Workshop*, December 1996.

[35] J. Rushby. *Safe and Secure Computing Systems*, chapter Kernels for Safety?, pages 210–220. Blackwell Scientific Publications, 1989.

[36] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.

[37] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1992.

[38] Richard N. Taylor. A general purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[39] A. Terry, R. London, G. Papanagopoulos, and M. Devito. The ARDEC/Teknowledge architecture description language (ArTek). Technical report, Teknowledge Federal Syst. and U.S. Army Armament Research, Development, and Eng. Center, July 1995. Version 4.0.

[40] Jeffrey M. Thompson. NIMBUS: A framework for static analysis and simulation of system-level inter-component communication. Master's thesis, University of Minnesota, December 1999.

[41] Jeffrey M. Thompson, Mats P.E. Heimdahl, and Steven P. Miller. Specification based prototyping for embedded systems. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, number 1687 in LNCS, pages 163–179, September 1999.

[42] W. Tracz. LILEANNA: A parameterized programming language. In *Proceedings of the Second International Workshop on Software Reuse*, pages 66–78, Lucca, Italy, March 1993.

[43] S. Vestal. MetaH programmer's manual. Technical report, Honeywell Technology Center, Minneapolis, MN, April 1996. Version 1.09.

[44] Michael W. Whalen. A formal semantics for RSML^{-e}. Master's thesis, University of Minnesota, May 2000.

[45] K.G. Wika. *Saefty Kernel Enforcement of Software Safety Policies*. PhD thesis, University of Virginia, May 1995.

[46] K.G. Wika and J.C. Knight. On the enforcement of software safety policies. In *Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS'95)*, pages 83–93, 1995.

[47] Michal Young. How to leave out details: Error-preserving abstractions of state-space models. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis (TAV 2)*, 1988.