The NIMBUS Environment
for
Specification of Safety Critical Systems
DRAFT

For the RSML-e Specification Language

Developed by

The Critical Systems Research Group (CriSys)
Department of Computer Science and Engineering
University of Minnesota
Minneapolis, Minnesota

Manual By
Mike W. Whalen, Mats P.E. Heimdahl, and Jeffrey M. Thompson,
# Table of Contents

1. Introduction and Document Overview.............................................................................................. 5  
   1.1 Constructing Specifications ........................................................................................................ 5  
   1.2 Simulating and Analyzing Specifications.................................................................................. 5  
   1.3 Examples................................................................................................................................... 6  
   1.4 Technical Documentation and Research Results....................................................................... 6  

2. Getting Started with NIMBUS ............................................................................................................ 7  
   2.1 System Requirements ............................................................................................................... 7  
   2.2 Installation and Setup Instructions .......................................................................................... 7  
      2.2.1 Acquiring the Installation Files.......................................................................................... 7  
      2.2.2 Installing the Tools.......................................................................................................... 7  
   2.3 Tools of the NIMBUS Environment .......................................................................................... 8  
      2.3.1 NimbusSim...................................................................................................................... 8  
      2.3.2 NIMBUS Manager............................................................................................................. 8  
      2.3.3 NIMBUSChannel Client................................................................................................. 8  

3. Introduction to RSML                                                                                   9  
   3.1 Synchronous Languages ....................................................................................................... 11  
   3.2 Variables and States Variables ............................................................................................ 13  
   3.3 Types..................................................................................................................................... 14  
   3.4 Variables................................................................................................................................ 14  
   3.5 Input Variables................................................................................................................... 19  
   3.6 Interfaces........................................................................................................................... 20  
   3.7 Expressions in RSML........................................................................................................... 26  
   3.8 Macros and Functions.......................................................................................................... 27  
   3.9 Advanced Language Issues.................................................................................................. 30  
      3.9.1 Circular Dependencies.................................................................................................. 30  
      3.9.2 Transition Issues and Equivalence Class Evaluation.................................................. 30  
      3.9.3 Receive Handlers......................................................................................................... 30  
      3.9.4 PREV_VALUE and PREV_STEP................................................................................ 31
1 Introduction and Document Overview

The Critical Systems Research Group originally developed the NIMBUS environment for safety critical systems at the University of Minnesota. The environment provided a framework for the development of software for safety critical systems, including simulation, code generation, and visualization. It supported the RSML (Requirements State Machine Language) formalism. In cooperation with Safeware Engineering Corporation, the existing tools were modified to support the RSML-e language and expanded to include a graphical user interface for the simulation engine, a construction tool for the specifications, and implementation of analysis.

This document contains information necessary to be productive with the NIMBUS environment.

1.1 Constructing Specifications

Specifications in the Nimbus environment are simple ASCII test files that can be created with any text editor. We are currently working on support for creating RSML-e specifications in the WinEdt editor. This support, however, is not available at this time. When the specification is ready, it is readable by the NimbusSim tool and can be simulated or analyzed. The following document describes the way in which specifications are constructed in the Nimbus environment.

- RSML-e Language Manual: The language manual provides a basic introduction to the syntax and semantics of the RSML-e language. Through the use of a simple example, the pump control system, the reader will gain an understanding of the features of the language and how they can be used.

1.2 Simulating and Analyzing Specifications

The NimbusSim tools provide the Nimbus environment with the ability to simulate and analyze specifications. The NimbusSim application itself provides the ability to load and graphically display specifications. Furthermore, the analyst can execute and analyze the specification. In addition to the NimbusSim simulator, the Nimbus environment provides for inter-process communication to be used for input and output of the simulations. This feature is supported by several applications: the NimbusManager and COM objects for integration to C++ and Visual Basic, and other associated clients. This framework allows the RSML-e specification to be executed with a variety of other components that accurately model the environment of the controller. All this is described in the simulation and analysis documentation:

- NimbusSim Graphical User Interface Manual: This document describes the features of the NimbusSim graphical user interface menus, toolbars, and command window. It discusses how to load specifications, layout the state hierarchy, simulate specifications, connect channels, and perform analysis.

- NimbusChannel Users Guide: This guide describes how NimbusSim can be connected to other applications via the NimbusChannel framework built on top of Microsoft's Component Object Model (COM). It discusses both which clients are currently available and how to create clients in C++ or Visual Basic using the provided COM objects and wrapper classes.
1.3 Examples

Examples are an important part of learning any new specification language. Research is ongoing at the University of Minnesota and elsewhere into how to structure and construct specifications in the RSML-e language. Even so, we provide here one completely worked-out example showing various features of the language and how they can be used.

- **The Clean Room:** Consider a room that is supposed to be sealed at all times. To enter the room you have to go through an airlock. To get in, you have to open the front door, step into the air-lock, close the door, open the inside door, step into the room, and finally close the inside door. This simple example will be used to illustrate many concepts from the RSML-e language.

Additional examples will be added as time progresses.

1.4 Technical Documentation and Research Results

Any documentation for the RSML-e language and its associated tools would be incomplete without a discussion of the formal semantics (a major feature of the language) as well as the research results that led to the creation of RSML-e.

These documents are currently available from the Critical Systems research groups at the University of Minnesota.
2 Getting Started with NIMBUS

NIMBUS is a powerful and flexible framework for the construction of specifications for safety-critical systems that is, nonetheless, easy to use. To get started, read the release notes provided with the installation files. These notes contain important information about late-breaking features, compatibility and installation issues, and more.

2.1 System Requirements

The NIMBUS environment runs under Windows 95, Windows 98, Windows NT 4.0, and Windows 2000. To take full advantage of the environment, users should have the following software (not provided by the University of Minnesota) installed on their workstations:

- Microsoft Internet Explorer 4.0 or higher (required by HTML help)

Nevertheless, NIMBUS can be used without this additional software; the user will simply not be able to utilize the features of the HTML format help (HTML help can be browsed with any web browser).

2.2 Installation and Setup Instructions

2.2.1 Acquiring the Installation Files

The installation files for the NIMBUS environment are downloadable from the Critical Systems Research Group's web site at the following URL. You must obtain a username and password from either the Critical Systems Research Group to successfully download the distribution. The URL for the download is:

http://www.cs.umn.edu/crisys/NIMBUS/download

Follow the instructions given on the page to complete the download of the NIMBUS Environment installation files.

2.2.2 Installing the Tools

To install the tools, follow the simple steps below:

1. Download the installation file from the web (or from the CD)
2. Unzip the installation file into a temporary directory.
3. Double click on the NIMBUS Installer program file
4. Follow the instructions through the installation
Important Note: If you already have a previous version of NIMBUS installed on your machine, you must first uninstall NIMBUS before you can attempt a new install. We do not know the source of the problems, but the installation program (provided by Microsoft) tends to freeze up or exhibits other kinds of unexpected behaviors if installing over an existing version is attempted. Thus, **uninstall first and then install a new version.**

2.3 Tools of the NIMBUS Environment

The NIMBUS environment encompasses a number of different tools. Thus, a number of items are contained in the start menu program group that is created by the setup program. The paragraphs below describe the tools that are available in the environment, their function, and where the documentation about their features can be located.

2.3.1 NimbusSim

NIMBUSim is the full-featured simulation and analysis tool of the NIMBUS framework. It allows you to perform simulation and analysis on RSML-e specifications, copy and paste specification layouts, produce testing scripts and more. NIMBUSim can be accessed with the NIMBUSim icon from the program group created by setup.

Documentation of NIMBUSim can be found in the following places:
- Application specific topics are found in the online help
- The NIMBUSim Graphical User Interface Manual

2.3.2 NIMBUS Manager

The NIMBUS Manager is the application that allows you to control the connections between various components in the NIMBUSChannel communication environment. The NIMBUSManager allows you to connect and disconnect sources and destinations so that the correct systems configuration can be achieved. The NIMBUS Manager can be accessed with the NIMBUS Manager icon from the program group created by setup. Documentation of the NIMBUS Manager can be found in the NIMBUSChannel Users Guide.

2.3.3 NIMBUSChannel Client

The NIMBUSChannel Client is a simple client application written in C++ that allows the user to connect to the NIMBUSChannel framework and act as a source, destination, or observer on a channel. The NIMBUSChannel client can be accessed through the program group created by setup by selecting the NIMBUSChannel Client icon. Documentation for the NIMBUSChannel Client can be found in the NIMBUSChannel Users Guide.
3 Introduction to RSML\textsuperscript{e}

RSML\textsuperscript{e} is a finite-state machine based specification language that represents the evolution of the specification language RSML (Requirements State Machine Language). RSML was designed during the specification of TCAS II (Traffic Alert and Collision Avoidance System II) by the Safety research group at the University of California, Irvine. RSML was based on Statecharts, including such features as the event propagation mechanism and the notion of hierarchical state machines.

RSML\textsuperscript{e}, a refinement and improvement of RSML, was created for a number of reasons. Among them were the error prone use of the event propagation mechanism and the difficulty of reusability in RSML specifications.

The purpose of RSML\textsuperscript{e}, and the associated tools in the Nimbus environment, is to allow analysts to specify safety critical systems with high reliability. RSML\textsuperscript{e} contributes to this goal by being a readable specification language that is usable and understandable by all stakeholders in a specification effort. Therefore, the language is suitable for manual inspections and reviews. Nevertheless, RSML\textsuperscript{e} is a fully formal specification language; thus, analysts can also perform formal analysis and simulation on the requirement model. To achieve a high level of confidence, all three approaches (manual inspections, formal analysis, and simulation) must be used in concert. This is enabled in the Nimbus environment by the RSML\textsuperscript{e} language and its associated tool set.

An RSML\textsuperscript{e} specification consists of a collection of variables, the next state relations for the variables, functions, macros, constants, and interfaces. Variables describe the internal state of the system. Interfaces describe how the specification interacts with the external environment. Functions and Macros are mechanisms for representing common expressions and predicates, respectively, in order to make specifications more concise. These constructs are discussed in the following sections with the context of a simple example: the clean room.

The clean room specification is as follows:

\begin{quote}
Consider a room that is supposed to be sealed at all times. To enter the room you have to go through an airlock. To get in, you have to open the front door, step into the air-lock, close the door, open the inside door, step into the room, and finally close the inside door. To open a door, a person must request the door using some means (e.g. a button). Only one person should be allowed in the airlock at a time, and if the airlock is in use, other requests should be denied until the airlock is unoccupied. At no point should both doors to the airlock be open, unless a power failure or catastrophic event occurs. If both doors are open, then the clean room must be considered contaminated, with serious financial consequences.

When entering the clean room, an individual must be “cleaned” using air scrubbers to remove particles from their clothes. The duration for this cleaning is some application-
\end{quote}
defined constant. Until the individual is clean, they should not be allowed into the clean room.

The system shall provide two alarm features. If an airlock is occupied for longer than a specified duration, a timeout alarm should be generated. Also, in case of some system malfunction or other catastrophe, pressing buttons within the clean room and the airlock will generate a panic alarm. In this event, both doors should be unlocked and people should be able to leave the clean room unhindered. If an alarm is generated, it continues until an administrator resets the system.

The clean room may have 1-n airlocks, all of which behave identically.

In this manual, we have made a few simplifying assumptions to the clean room problem:

- The clean room only contains one airlock
- It is the administrator’s responsibility to ensure that the system is in a consistent state when the system is reset (i.e. no people in the airlocks)
- The sensors/actuators do not malfunction
- The cleaning interval is 60 seconds; the timeout interval is 5 minutes

Here are the system inputs/outputs.

<table>
<thead>
<tr>
<th>Input:</th>
<th>Description:</th>
</tr>
</thead>
<tbody>
<tr>
<td>panic_button: bool</td>
<td>panic_button is true when any of the panic buttons are pressed</td>
</tr>
<tr>
<td>reset_button: bool</td>
<td>reset_button is true when a system reset request is generated</td>
</tr>
<tr>
<td>inner_door_request: bool</td>
<td>inner_door_request is true in the duration when a user is requesting to exit the clean room</td>
</tr>
<tr>
<td>outer_door_request: bool</td>
<td>outer_door_request is true in the duration when a user is requesting to enter the clean room</td>
</tr>
<tr>
<td>inner_door_open: bool</td>
<td>inner_door_open is true when the inner door is open</td>
</tr>
<tr>
<td>outer_door_open: bool</td>
<td>outer_door_open is true when the outer door is open</td>
</tr>
<tr>
<td>airlock_occupied: bool</td>
<td>airlock_occupied is true when the airlock is occupied (could be through a floor or motion sensor)</td>
</tr>
<tr>
<td>clock: bool</td>
<td>A clock pulse that is issued once per second</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output:</th>
<th>Description:</th>
</tr>
</thead>
<tbody>
<tr>
<td>inner_door_lock</td>
<td>inner_door_lock is true when the inner door of the airlock is locked</td>
</tr>
<tr>
<td>outer_door_lock</td>
<td>outer_door_lock is true when the outer door of the airlock is locked</td>
</tr>
<tr>
<td>decontaminate: bool</td>
<td>decontaminate is true during the decontamination interval when the system should “scrub” a user who is entering the clean room</td>
</tr>
<tr>
<td>panic_alarm: bool</td>
<td>panic_alarm is true at the instant when a user presses the panic_button, and true thereafter until the reset_button has been pressed</td>
</tr>
</tbody>
</table>
timeout_alarm: bool  timeout_alarm is true at the instant when the user has been in an airlock for longer than 300 seconds

Once these inputs are set, it is pretty straightforward to describe the procedure for entering/exiting the airlock:

1. Initially, both doors to the airlock are locked.
2. The user requests to enter (exit) the airlock
   - If the airlock is ‘in-use’ the request is denied
   - Otherwise, unlock the outer (inner) door
3. The user opens the outer (inner) door
4. The user closes the outer (inner) door
   - If the airlock is occupied, then lock the outer (inner) door and proceed to the next stage
   - If the airlock is unoccupied, the user must have decided not to enter the airlock, so the airlock is no longer ‘in-use’
5. If the user is entering, then clean the user for 60 seconds
6. Unlock the inner (outer) door and wait for the user to exit
7. The user opens the door
8. The user closes the door
   - If the airlock is still occupied, then the user must not yet have exited; repeat
   - If the airlock is unoccupied, then the process is complete and the airlock is no longer ‘in-use’.

### 3.1 Synchronous Languages

The semantics of RSML\textsuperscript{-e} puts it in a class of languages called synchronous languages. Since this class of languages may be largely unknown to the novice RSML\textsuperscript{-e} user, we here include a short introduction.

Synchronous languages were proposed as a software engineering tool in the late 1970s and independently in the programming language community in the late 1980s as a technique for modeling and constructing reactive (i.e. process control) systems. As these systems often control the behavior of several cooperating machines, they are often most easily modeled as a set of cooperating concurrent tasks. Synchronous languages provide primitives that allow programmers to naturally model this concurrent structure, and also to consider that their programs react *instantaneously* to external events [Halbwachs91]. By using this *synchrony hypothesis*, and enforcing certain constraints within a synchronous language, it is possible to create logically concurrent programs that are deterministic both in terms of functionality and time.
There are two main styles of synchronous languages: imperative and dataflow. Imperative synchronous languages are quite similar to standard imperative languages like C or Ada, but are augmented with constructs to deal with process instantiation, communication, and termination. Unlike similar constructs in, for example, Ada, these constructs are considered logically instantaneous, and are essential to creating deterministic programs. The leading example of this style is the programming language Esterel[Berry00].

In contrast, dataflow languages are inspired by control theory. Many of the engineers who design reactive systems model their systems as networks of operators transforming flows of data, and at a higher level by block diagrams that group these networks into reusable components. Dataflow languages allow these models to directly realize the software control system.

As the basis of a high-level programming language, the dataflow model has several merits:

- It is a completely functional model without side effects. This feature makes the model well suited to formal verification and program transformation. It also facilitates reuse, as a module will behave the same way in any context into which it is embedded.

- It is a naturally parallel model, in which the only constraints on parallelism are enforced by the data-dependencies between variables. This allows for parallel implementations to be realized, either in software, or directly in hardware.

Dataflow models can be either synchronous or asynchronous. In an asynchronous dataflow model, the outputs of the system are continually recomputed depending on the inputs to the system. In the synchronous model, however, real-time is broken into a sequence of instants in which the model is recomputed. The synchronous model is better suited to translation into a programming language, as it more naturally matches the behavior of a computer program. Therefore, all of the dataflow-style languages adopt some form of this approach.

As an example, consider a system that computes the values of two variables, X and Y, based on 4 inputs: a, b, c, and d:

\[
\begin{align*}
X &= 2a / (b - c) \\
Y &= X + d
\end{align*}
\]

This diagram is to be read left-to-right, with the inputs "flowing" through the system of operators to create the outputs at the right side. The diagram can be represented more concisely as a set of equations, as shown at right. We name the inputs to the dataflow model input variables and all variables that are computed by the model state variables.
The variables in a dataflow model are used to label a particular computation graph; they are not used as constraints. Therefore, it is incorrect to view the equations as a set of constraints on the model: a set of equations such as \{X = 2a/Y, \ Y = X + d\} does not correspond to an operator network because X and Y mutually refer to one another. Such a system may have no solution or infinitely many solutions, so cannot be directly used as a deterministic program. If viewed as a graph, these sets of equations have data dependency cycles, and are considered incorrect.

However, in order for the language to be useful, we must be able to have mutual reference between variables. To allow benign cyclic dependencies, a delay operator is added. The operator returns the value of an expression, delayed one instant. For example: \{X = 2a / Y, \ Y = \text{delay}(X, 1) + d\} defines a system where X is equal to 2a divided by the current value of Y, while Y is equal to the previous value of X plus the current value of d. The second parameter of the delay operator defines the value of the operator at the initial instant, when the previous value of X is undefined. Systems of equations of this form always have a single solution. The delay operator is also the mechanism for recording state about the model. For example, we can construct a counter over the natural numbers by simply defining the set of equations \{x := \text{delay}(x+1, 0)\}.

Finally, some notion of selection is added to assignment expressions. Depending on the language, this notion can be realized as an if/then statement, a series of cases, or a set of transitions. From these elements, at its core, a dataflow program can be viewed as simply a set of input variables and assignment equations of the form \{X_0 = E_0, \ X_1 = E_1, ..., \ X_n = E_n\} that must be acyclic in terms of data dependencies.

RSML$^e$ is a synchronous data-flow language of the structure described above.

### 3.2 Variables and States

Variables are central to RSML$^e$. An RSML$^e$ specification is constructed from variables organized in parallel or hierarchically (this will be explained in more detail below). We make a slight distinction between variables and state variables. State variables in RSML$^e$ are simply variables that have graphical representation that will show up in the GUI during execution. In the remainder of this report we will use variables and state variables interchangeably—they are both referring to some variable in the model that is used to capture the system state.

In other specification languages, states are often used to represent activities of the controller. This is a subtle, yet important, difference; the state of the system may or may not have anything to do with which particular activities the system is performing at any particular moment. Therefore, in RSML$^e$ states can be thought of as more of a hierarchical enumerated type and are closer to traditional finite state machines than those found in other specification languages.

Variable assignments in RSML$^e$ govern how the state variables can change from one value to another—they capture the next state relation. There are two styles of variable assignments—the transition style and the assignment style. Both styles will be illustrated below.
### 3.3 Types

The various variables in an RSML specification must have a type. Thus, the types needed to model a system are often determined early during modeling. In this case, we will model open and closed doors, alarms that may be on or off, and doors that can be locked or unlocked. For the clean room, we start with the types below

```plaintext
TYPE_DEF on_off { off, on }
TYPE_DEF door_status { closed, open }
TYPE_DEF door_lock_status { unlocked, locked }
TYPE_DEF button_status { not_pressed, pressed }
```

Type definitions in RSML simply declare a user defined enumerated type. Note here that all types in RSML contain an implicit value UNDEFINED. This value is used when we simply do not know what the value of a variable may be. This situation occurs, for example, at startup, when no input arrives, or when the variable is not relevant (the part of the variable hierarchy where it resided is not used at the time). The issue of UNDEFINED will be revisited later in this manual.

The grammar for type definitions is very simple.

```plaintext
type_def : TYPE_DEF IDENTIFIER '{' enum_element_list '}'

enum_element_list : IDENTIFIER
| enum_element_list ',' IDENTIFIER
```

### 3.4 Variables

When a variable is declared, we have to give it a type. This is achieved through a type reference or an implicit type declaration.

```plaintext
type_ref : IDENTIFIER
| INTEGER_TYPE
| REAL_TYPE
| BOOLEAN_TYPE
| TIME
```

The clean room specification is straightforward to implement in RSML. Using the RSML hierarchy constructs (discussed below), it is possible to directly visualize and control the relationships between the variables that describe the status of the airlock:
At the top level, we have an `airlock_status` variable, which describes whether or not the airlock is occupied, along with variables detailing the status of alarms and door locks. Under the `airlock_status` variable, we place the variables describing the process of entering and exiting the airlock. These variables are only relevant when `airlock_status` is in entry or exit mode, respectively.

This view of the state variables is the one produced by the NIMBUSSim RSML$^e$ simulator. It shows each variable value as a box with black text and a white background (this is the default in the tool—it can be customized by the user). The possible values of a variable are joined with connecting lines. Note, however, that these lines do not necessarily indicate that it is possible for the state variable to change from any of its values to any other of its values (see below, about assignment relations). **Note:** This visualization may change since CriSys are not quite satisfied with the information content. The name of state variable appears above the connected states. State variables can be nested (as shown by the `airlock_status` variable).
When an RSML specification is loaded into the NimbusSim simulator, the simulator needs to assign some initial values to the state variables. This initial value is determined by which states are marked with the INITIAL_VALUE keyword in the proceeding state definitions. Thus, Figure 2 above shows the state hierarchy for the pump controller in the default configuration.

The collection of all the active states in the state hierarchy can be thought of as the configuration of the machine. The behavior of the specification is determined, in part, by the way that the state hierarchy changes from one configuration to another. These changes in configurations are determined by the definition of transitions from one state to another. Transitions are specified one state variable at a time.

Consider the behavior of the airlock_status state variable. This state variable is intended to capture the state of the airlock (is a person entering the room, exiting the room, or is the lock unoccupied). The definition of the state variable can be seen in Figure 3 below.

```plaintext
STATE_VARIABLE airlock_status :
  VALUES : {unoccupied, entry, exit}
  PARENT : None
  INITIAL_VALUE : unoccupied
  CLASSIFICATION: State
  Transition unoccupied TO exit IF inner_door_request
  Transition unoccupied TO entry IF
    TABLE
      outer_door_request : T;
      inner_door_request : F;
    END TABLE
  Transition exit TO unoccupied IF
    TABLE
      PREV_STEP(..airlock_exit) IN_STATE completed : T *;
      reset_button : * T;
    END TABLE
  Transition entry TO unoccupied IF
    TABLE
      PREV_STEP(..airlock_entry) IN_STATE completed : T *;
      reset_button : * T;
    END TABLE
END STATE_VARIABLE
```

Figure 3: The definition of the airlock_status state variable.

The definition includes the name of the state variable, airlock_status and the possible values the variable can take on (note that all variables can take on the value UNDEFINED in addition to
any values defined in the variable definition—this will be discussed in detail later). The next thing in the definition is the parent of the variable. This is the construct that allows us to organize variables in a hierarchy. For example, the variable airlock_entry has the parent airlock_status.entry (the value entry of airlock_status). The variable airlock_entry has no parent (it is on the top level) and this is indicated by the parent None. The CLASSIFICATION field of the variable definition is intended for suture extensions of the language. Currently, the field is ignored unless it states that this is a State variable. All variables classified as State variables will be visualized in the graphical user interface. Next in the definition, the possible values for the state variable are given as well as the conditions that must hold for the state variable to assume the value. Essentially, each one of the statements represents a transition from one state (state variable value) to another state. Note that some of the conditions in the specification are presented in a DNF form called And/Or tables.

During their work on the TCAS specification, the Irvine group discovered that the traditional predicate logic statements traditionally used to capture the conditions on transitions did not scale well to the large, complex expressions in TCAS. To combat this issue, they developed And/Or tables. The tables contain a column of predicates. To the right of the column of predicates are one or more columns of truth-values. During evaluation (execution of the specification), the predicates at the left will have some values. For a column in the table to be TRUE, the values of the predicates must “match up” with the truth-values indicated in one the columns (i.e., the predicate must the TRUE if there is a T in the column and FALSE if there is an F). A * in the column denotes that we don’t care about the value of that particular predicate with respect to the column. For an And/Or table to be TRUE, one of the columns must completely match (i.e., be TRUE).

The conditions in the airlock_status state variable are relatively simple. Inner_door_request is a Boolean input variable that is used as a condition. We can also use abbreviations of more complex conditions (called macros) and various expressions involving integer and real variables, and time. A complete discussion of all the expressions allowed in RSML$^{e}$ appears later in the document.

In some instances we may want a state variable to take on a specific value independently of what value it had before—in essence we want transitions from ANY state to a certain state. In RSML$^{e}$ we achieve this with a slightly different variant of the transition definition. As can be seen in Figure 4, we are allowed to state that a variable assumes a value when a condition is true (independently of what value it had previously).

```plaintext
STATE_VARIABLE inner_door_lock : door_lock_status
    PARENT: NONE
    INITIAL_VALUE : locked
    CLASSIFICATION : State
    EQUALS unlocked IF inner_door_unlocked()
    EQUALS locked IF !inner_door_unlocked()
END STATE_VARIABLE
```

Figure 4: The EQUALS style of transition definition.
The grammar for state variable definitions is given below.

```plaintext
state_variable_def : STATE_VARIABLE IDENTIFIER array_decl ':'
                   PARENT ':' parent_decl
                   INITIAL_VALUE ':' expression
                   variable_numeric_decl
                   classification_def
                   case_list
                   END STATE_VARIABLE
                   ;

variable_numeric_decl : /* empty */
                       | UNITS ':' IDENTIFIER
                       | EXPECTED_MIN ':' expression
                       | EXPECTED_MAX ':' expression

variable_type_decl : type_ref
                   | VALUES ':' '{' enum_element_list '}'

array_decl : /* empty */
            | '[' expression TO expression ']

parent_decl : NONE
             | parent_name_path
             ;

parent_name_path : IDENTIFIER
                  | parent_name_path '.' IDENTIFIER
                  ;

classification_def : /* empty */
                    | CLASSIFICATION ':' IDENTIFIER
                    ;

type_ref : IDENTIFIER
          | INTEGER_TYPE
          | REAL_TYPE
          | BOOLEAN_TYPE
          | TIME
          ;

classification : /* empty */
                | CLASSIFICATION ':' IDENTIFIER
                ;

condition : TABLE
           row_list
           END TABLE
```

Introduction to RSML$^c$ and Nimbus
expression /* Must return BOOLEAN */
;
row_list : expression ':' truth_value_list ';'
  | row_list expression ':' truth_value_list
  | ';

truth_value_list : truth_value
  | truth_value truth_value_list
  |

truth_value : 'T'
  | 'F'
  | '.'
  | '*'

3.5 Input Variables

Input variables in RSML\textsuperscript{e} allow you to model information about the environment of the system. For example, you might like to capture the inputs from sensors. The input variables are conceptually slightly different than the variables we discussed in the previous section—input variables can only be set by an interface when an interaction with the environment occurs. There is not notion of an output variable in RSML\textsuperscript{e}; any variable can be used to output information from a model. We do, however, encourage the use of dedicated variables that act as output variables. Variables (both input variables discussed here and the ‘normal’ variables discussed in the previous section) can be any one of the types that are present in the RSML\textsuperscript{e} type system: floating point, integer, time, or enumerated.

A type of a variable (its possible values) can be declared separately in RSML\textsuperscript{e}, or it can be declared directly with the variable itself. The latter is called using an anonymous type since the type is never given a name and cannot be used in any other place of the specification. The other alternative is to declare a type explicitly and then use the type name when we declare the type of a variable (Figure 4).

An input variable consists of an initial value, and (for non-enumed type variables) an expected minimum, maximum, and units. The definition of the panic_button from the clean room is shown in Figure 5. The type of the variable is Boolean. The definition of the panic_button does not require an expected minimum, expected maximum, or units since it is Boolean. Only variables of numeric types (integer and real) require those fields. There are no numeric variables in the clean room example—an example of a numeric variable from another specification (an avionics system) is included in Figure 6 for illustration.
3.6 Interfaces

Interfaces define a number of properties related to how the specification can interact with its environment. In RSML-e the interaction with the environment is achieved by providing or consuming messages over a communication channel. Since RSML-e is a synchronous language, we make the assumption that only one message is received at any point in time and that the message can be completely processed before another one arrives. If we want to model that many inputs can change simultaneously, we can put them all in the same message—the fields in a message can all change at the same time, but the fields in different messages cannot. Thus, RSML-e operates under a one-message assumption. An example of a message definition in the clean room is shown in Figure 7. A different message from an avionics application is shown in Figure 8.

MESSAGE Update_Message {
    f_panic_button IS boolean,
    f_reset_button IS boolean,
    f_inner_door_request IS boolean,
    f_outer_door_request IS boolean,
    f_inner_door_open IS boolean,
    f_outer_door_open IS boolean,
    f_airlock_occupied IS boolean
}

Figure 7: Message definition in the clean room.
MESSAGE AltitudeMessage {
    Alt IS INTEGER,
    aq IS AltitudeQualityType
}

**Figure 8: Message definition for altitude.**

First, the interface defines the type of communication: Send-Receive or Publish-Read. Send-Receive communication is equivalent to message passing scheme. Publish-Read communication, on the other hand, buffers the message on the channel so that a message that is published might be read several times by the reader. Second, interfaces define the properties of the communication, for example the expected minimum and maximum separation between messages over the channel. Finally, the interfaces regulate the assignment of inputs to the specification to the input variables. Using this feature, simple safety and liveness constraints can be imposed by the interfaces without considering the (potentially complex) function represented by the state machine and associated definitions.

The interface for the clean room is given below (Figure 9). The interface definition begins with a declaration of the name for this interface. The simulator uses this name to hook the interface to actual communications channels from the environment. Next, the expected minimum and maximum separation for messages over the channel is given. The final part of the header for the input interface is the input action. This determines (1) the type of communication and (2) which message will be received/read on the input channel.

```plaintext
IN_INTERFACE Update_Interface :
    MIN_SEP : UNDEFINED
    MAX_SEP : UNDEFINED
    INPUT_ACTION : RECEIVE(Update_Message)
    HANDLER :
        CONDITION : TRUE
        ASSIGNMENT
            panic_button := f_panic_button,
            reset_button := f_reset_button,
            inner_door_request := f_inner_door_request,
            outer_door_request := f_outer_door_request,
            inner_door_open := f_inner_door_open,
            outer_door_open := f_outer_door_open,
            airlock_occupied := f_airlock_occupied
        END ASSIGNMENT
    END HANDLER
END IN_INTERFACE
```

**Figure 9: Interface definition for the clean room.**

The final section of the input interface consists of a number of handlers. Handlers are similar to transitions in a state variable. They have a condition, under which they will be executed. If the condition is TRUE, then the handler will execute and the assignments will occur. In the case in
Figure 9, the interface only has one handler and that handler will always be used (the guarding condition for the handler is simply true). A more complex interface from an avionics example is included in Figure 10.

```
IN_INTERFACE AltitudeMessageInterface :
  MIN_SEP : 50 MS
  MAX_SEP : 100 MS
  INPUT_ACTION : RECEIVE(AltitudeMessage)

RECEIVE_HANDLER :
  CONDITION :
    TABLE
      Alt <= Altitude::EXPECTED_MAX : T;
      Alt >= Altitude::EXPECTED_MIN : T;
    END TABLE
  ASSIGNMENT
    Altitude := Alt,
    AltitudeQuality := aq
  END ASSIGNMENT
END HANDLER

RECEIVE_HANDLER :
  CONDITION :
    TABLE
      Alt <= Altitude::EXPECTED_MAX : F *
      Alt >= Altitude::EXPECTED_MIN : * F
    END TABLE
  ASSIGNMENT
    Altitude := UNDEFINED,
    AltitudeQuality := Bad
  END ASSIGNMENT
END HANDLER

END IN_INTERFACE
```

Figure 10: More complex interface example from avionics.
The BNF grammar for input interfaces follows:

```
in_interface_def : IN_INTERFACE IDENTIFIER ':'
                  MIN_SEP ':' expression
                  MAX_SEP ':' expression
                  INPUT_ACTION ':' in_interface_type_spec
                  '(' IDENTIFIER ')'
                  in_handler_list
                  END IN_INTERFACE

in_interface_type_spec : RECEIVE
                      | READ

in_handler_list : in_handler
                | in_handler in_handler_list

in_handler : in_handler_type ':'
            CONDITION ':' condition
            in_assignment
            END HANDLER

in_handler_type : RECEIVE_HANDLER
                | HANDLER

in_assignment : /* empty */
               | ASSIGNMENT
               in_assignment_list
               END ASSIGNMENT

in_assignment_list : identifier_name_path ASSIGN_TOKEN expression
                   | in_assignment_list ',' identifier_name_path
                   ASSIGN_TOKEN expression
```

```
```

Introduction to RSML-e and Nimbus
Output interfaces are similar to input interfaces. An example of an output interface from the pump controller is given in Figure 11. Again, a more complex example from avionics is included in Figure 12.

```rsml
OUT_INTERFACE Actuator_Interface :
    MIN_SEP : UNDEFINED
    MAX_SEP : UNDEFINED
    OUTPUT_ACTION : SEND(Actuator_Message)
    HANDLER :
        CONDITION : TRUE
        ASSIGNMENT
            f_inner_door_lock := inner_door_lock,
            f_outer_door_lock := outer_door_lock,
            f_decontaminate := decontaminate,
            f_panic_alarm := panic_alarm,
            f_timeout_alarm := timeout_alarm
        END ASSIGNMENT
        ACTION : SEND
    END HANDLER
END OUT_INTERFACE
```

**Figure 11: Simple output interface.**

```rsml
OUT_INTERFACE FaultDetectionInterface :
    MIN_SEP : 50 MS
    MAX_SEP : 200 MS
    OUTPUT_ACTION : SEND(FaultMessage)
    HANDLER :
        CONDITION :
            TABLE
                ASWOpModes IN_STATE OK : T * ;
                ASWOpModes IN_STATE FailureDetected : * T;
            END TABLE
        ASSIGNMENT
            fault := FaultDetectedVariable
        END ASSIGNMENT
        ACTION : SEND
    END HANDLER
END OUT_INTERFACE
```

**Figure 12: More complex output interface from the avionics domain.**
Finally, we provide the BNF grammar for the output interfaces.

```
out_interface_def : OUT_INTERFACE IDENTIFIER ':'
                  MIN_SEP ':' expression
                  MAX_SEP ':' expression
                  OUTPUT_ACTION ':'

out_interface_type_spec '( IDENTIFIER )'
                  output_handler_list
                  END OUT_INTERFACE

out_interface_type_spec : SEND
                      | PUBLISH
                      
output_handler_list : output_handler
                      | output_handler_list output_handler

output_handler : HANDLER ':'
                  CONDITION ':' condition
                  out_assignment
                  ACTION ':' out_handler_type
                  END HANDLER

out_handler_type : SEND
                  | PUBLISH
                  | NONE

out_assignment : /* empty */
                | ASSIGNMENT
                | out_assignment_list
                END ASSIGNMENT

out_assignment_list : IDENTIFIER ASSIGN_TOKEN expression
                    | IDENTIFIER ASSIGN_TOKEN expression ','
                    | out_assignment_list
```

Introduction to RSML-ε and Nimbus  
Page 25 of 76
### 3.7 Expressions in RSML⁻

This section provides a look at the expressions available in RSML⁻ and their meaning. RSML⁻ supports the standard arithmetic expressions (addition, subtraction, multiplication, and division) comparison operators (greater than, greater or equal, equal (=), not equal (!=), less than, less than or equal) as well as parenthesis for expression grouping and traditional logical not for Boolean expressions (NOT expression). These expressions can contain references to variables, constants, macros, and functions.

Literal values are allowed in RSML⁻. Floating point, integer and Boolean literals are given as expected. Enumerated literals are given as `type_name:enumeration_name`. Time values are given in the format `n [H | M | S | MS]` where `n` is an integer and H, M, S, MS stand for hours, minutes, seconds and milliseconds respectively. Any number of these clauses can be given separated by white space and an optional "AND;" thus "3 H 5 M" and "4 H 5 M and 3 S" are both valid time literals.

Also supported is a set of RSML⁻ specific expressions involving, for example, previous values of variables and time. These expressions are detailed in Table 1 on page 27. The format of the entries of the table is in a pseudo-BNF grammar style. Parts that appear in italics are references to other language definitions or expressions. Parts which appear in square brackets [] are either optional or represent a choice of several values. For example [a] means that part “a” is optional and [a|b] means choose between “a” and “b.”

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>`variable_name::[EXPECTED_MIN</td>
<td>Expected to the expected minimum or maximum of the referenced variable</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>PREV_VALUE(<code>variable_name [, n]</code>)</td>
<td>The <code>n</code>th previous value of the variable referenced before the current step started. The <code>n</code> is optional. For example, PREV_VALUE(x) is the value that the variable x had before it took on the value that it had at the beginning of current step.</td>
</tr>
<tr>
<td>PREV_STEP(<code>variable_name [, n]</code>)</td>
<td>The value that the variable had in the <code>n</code>th previous step. The <code>n</code> is optional. For example, PREV_STEP(x) is the value that variable x had at the end of the previous step (and the beginning of this step).</td>
</tr>
<tr>
<td>TIME</td>
<td>The current system time.</td>
</tr>
<tr>
<td>TIME(<code>variable_name</code>)</td>
<td>The time when the variable acquired the current value.</td>
</tr>
<tr>
<td>PREV_ASSIGN(<code>variable_name [, n]</code>)</td>
<td>The time when the referenced variable acquired its <code>n</code>th previous value.</td>
</tr>
</tbody>
</table>
TIME_CHANGED \( (\text{variable\_name}[,n]) \)  The time when the state variable given by \text{variable\_name} changed value. Note that this time may be different than when the variable last got assigned (it could have been assigned the same value).

\text{interface\_name}::\text{LAST\_IO}  The time that the interface referenced last performed I/O on the channel.

\[ \text{interface\_name}::[\text{MIN\_SEP} | \text{MAX\_SEP}] \] The minimum or maximum separation for the given interface.

\text{expression} \text{EQ\_ONE\_OF} \text{expression\_list}  True if the \text{expression} is equal to one of the comma separated expressions in the \text{expression\_list}.

\text{ASSIGNED(\text{variable\_name})}  True if the variable was assigned in this step.

\text{CHANGED(\text{variable\_name})}  True if the variable changed its value in this step.

\text{AT\_TRUE(\text{expression})}  True if the expression changed from false to true in the current step.

\text{AT\_FALSE(\text{expression})}  True if the expression changed from true to false in the current step.

\text{AT\_CHANGED(\text{expression})}  True if the expression changed from one truth-value to another truth-value in the current step.

<table>
<thead>
<tr>
<th>Table 1: Expressions in RSML(^e).</th>
</tr>
</thead>
</table>

### 3.8 Macros and Functions

The macros and functions of RSML\(^e\) complete the language by allowing the analyst to define commonly used computations in a modular way. This facilitates good structure and maintainability in the specification. Macros in RSML\(^e\) are simply a compact method of writing Boolean functions. Macros are represented as an And/Or table (thus, in a way, a macro is simply a named And/Or table). A sample macro from the clean room can be seen in Figure 13. A function from the avionics domain is shown in Figure 14.

MACRO \text{inner\_door\_unlocked}() :
  TABLE
    \text{..airlock\_entry IN\_ONE\_OF} \{\text{awaiting\_exit, exiting}\} : \text{T} * *;
    \text{..airlock\_exit IN\_STATE} \text{entering} : * \text{T} *;
    \text{panic\_alarm = on} : * * \text{T};
  END TABLE
END MACRO

**Figure 13: Macro example.**
TYPE_DEF Selected_Nav_Types { FMS, VOR, LOC }

FUNCTION Selected_Nav_Type(): Selected_Nav_Types
   EQUALS FMS IF Is_Selected_Nav_Source_FMS()
   EQUALS VOR IF TABLE
      VNR_Signal_Type = VOR : T;
      Is_Selected_Nav_Source_VNR() : T;
   END TABLE
   EQUALS LOC IF TABLE
      VNR_Signal_Type = LOC : T;
      Is_Selected_Nav_Source_VNR() : T;
   END TABLE
END FUNCTION

Figure 14: Function example.

In RSML-e, both Macros and functions can take parameters. Parameters allow the macros and
functions to be even more modular and allow the analyst to reuse common conditions in various
situations in the specification. For example, you might wish to have a macro to do pair wise
sensor failure analysis for an array of sensors. The macro would contain the conditions for
determining failure given two sensor inputs and the parameters would by the sensor inputs. The
clean room specification does not contain any parameterized macros or functions, so no example
from that specification is possible here; however, the function definition in Figure 15 illustrates
the simple maximum function.

FUNCTION Max(a IS INTEGER, b IS INTEGER): INTEGER
   EQUALS a IF a > b
   EQUALS b IF a <= b
END FUNCTION

Figure 15: Parameterized function.
The BNF for macros and functions follow below:

```plaintext
/*------------ Macro definitions --------------------------------*/

optional_formalParms : /* EMPTY */
| '(' formalParameterList ')' 

macro_def : MACRO IDENTIFIER optional_formalParms ':'
  condition 
  END MACRO 
;

/*------------ Function definitions -----------------------------*/

optional_expr_list : /* empty */
| expression_list 

function_def : FUNCTION IDENTIFIER '('
  formalParameterList ')' ':' type_ref 
  case_list 
  END FUNCTION 
| STUB_FUNCTION IDENTIFIER '('
  formalParameterList ')' ':' type_ref 
  optional_expr_list 
  END STUB_FUNCTION 
;

case_list : /* EMPTY */
| case_list case 
;

case : EQUALS expression IF condition 
| TRANSITION expression TO expression IF condition 
;

actualParameterList : /* empty */
| expression_list 
;

formalParameterList : /* empty */
| IDENTIFIER IS type_ref 
| formalParameterList ',' IDENTIFIER IS type_ref 
;
```

Introduction to RSML© and Nimbus
3.9 Advanced Language Issues

RSML\textsuperscript{e} is based on the idea that the various language definitions in the specification constitute a *mathematical relation* from inputs to outputs. The relation is constructed by computing the values of the various language items (state, input, and output variables, macros, and functions). This is an important concept in the formalism of RSML\textsuperscript{e} because it allows the opportunity to create various completeness and consistency criteria for the relation. However, this view of the semantics leads to a certain behavior of specifications written in the notation that might not be readily apparent to the newcomer. These are briefly discussed in the sections below. For more information, please refer to the formal semantics of RSML\textsuperscript{e}.

3.9.1 Circular Dependencies

When an entity in the specification changes, the semantics of the RSML\textsuperscript{e} language dictate that all entities that depend on the one that changed must be recomputed. Thus, the order of computation in the specification is determined by the data dependencies of the various language entities. This means that there can be no circular dependencies, because there would be no way to tell when to stop updating the values in the specification. That is, the data dependencies for all entities in the specification must form a directed acyclic graph.

Using expressions that reference the previous values of variables, however, will not cause circular dependencies. That is because this previous information is fixed at the start of the computation and remains constant throughout. Thus, formally, these previous values can be viewed as parts of the system state (or as additional inputs to the specification).

3.9.2 Transition Issues and Equivalence Class Evaluation

The semantics of RSML\textsuperscript{e} states that for each state variable there should be a maximum of one transition taken in each step between its immediate children. A side effect of these is that the state of the machine is recomputed on a variable-by-variable basis, not on a transition-by-transition basis. The data dependencies for a variable are computed by taking the union of the data dependencies of the transitions between the values of the variable (all the expressions in the variable definition are taken into account). Furthermore, variables with parent state values are defined to be data dependent on their parent variables in the variable hierarchy so that variables that are “higher” in the tree get evaluated before the child variables.

3.9.3 Receive Handlers

Usually, entities in the RSML\textsuperscript{e} specification are recomputed when some other entity changes. However, the handlers in the RECEIVE interfaces are a special case because, in general, they should only be recomputed when a message was actually received regardless of the data dependencies of the conditions in the handlers. Therefore, to denote this separate behavior those handlers are marked with \texttt{Receive_Handler} instead of just \texttt{Handler}.
3.9.4 PREV_VALUE and PREV_STEP

The PREV_VALUE and PREV_STEP expressions in RSML\textsuperscript{e} allow the user to access the previous values of variables and interfaces. This does not change the view of the specification as a function. Rather, these values are (necessarily) static and thus are simply viewed as additional parts of the system state.
4 NIMBUSSim Graphical User Interface

4.1 The NIMBUSSim Interface

![Image of NIMBUSSim GUI Main Window]

Figure 16: The NIMBUSSim GUI Main Window

4.2 Overview

The NIMBUSSim Graphical User Interface provides accurate and fast access to the functionality of the NIMBUS simulator. Specifications are visualized in two ways: a tree-structure of detailed information about the data, and a State Hierarchy Diagram that enhances the context of state relationships. Furthermore, the user may observe the effects of execution with the run-time information provided by Active State Highlighting, Variable Watch updates, and the Clock display.

Launching NIMBUSSim causes two windows to appear: the Graphical User Interface (Fig. 1) and the Command-Line Interpreter. There exists one simulator and these two windows are two methods for issuing commands. The user may switch between them at any time.

In the next few sections we present an overview of the NIMBUS commands that are accessible through the GUI. The first section links the GUI commands with their visual appearance. The
next section summarizes the steps required to run a simulation, as they pertain to the User Interface. Finally, the Command Reference summarizes the GUI and Command Line counterparts, with some additional actions that are unique to each.

4.3 Details

4.3.1 File Menu Options

Figure 17: The File Menu

Load Specification

Opens an existing specification causing it to be loaded into the simulator. Upon successful parsing, the state hierarchy is diagrammed and specification details are loaded into the Tree View.

Load Script File

Opens a script file and loads it into the simulator for immediate evaluation.

Options

Here you can access many of the user-configurable properties from the System Options dialog. In addition, most of these options are available through their associated toolbars and menus.

Exit

Exits the simulator and interface. If there is a simulation in progress, then you must Stop the Full Execution.
4.3.2 System Options

Figure 18 Layout Options

Layout options include orientation. Choose Horizontal for one that favors wide layouts. Choose Vertical for one that favors taller layouts. The Vertical layout is more useful for transferring the image to a word processor.

Figure 19 Color Options for State Hierarchy
You may select the State Hierarchy Diagram colors.

Figure 20 Preview Panel for State Hierarchy Options

You can preview the Layout, Orientation and Color configurations before committing to them.

Figure 21 Simulator Options

State Diagram Updating

Check or uncheck to either enable or disable the updating of the State Hierarchy Diagram during any run command executions.
**Watch Window Updating**

Check or uncheck to enable or disable the updating of the Watch Window during any run command executions.

**Real Time**

Set the simulation clock type to either Real Time (checked) or Simulation Time (unchecked).

### 4.3.3 View Menu

![The View Menu](image)

**Figure 22** The View Menu

**Main Toolbar**

Show (checked) or hide (unchecked) the Main Toolbar. This toolbar contains commands: Load Specification, Help

**State Diagram Toolbar**

Show (checked) or hide (unchecked) the State Diagram Toolbar. This toolbar contains commands: Load Specification, Help

**Simulation Toolbar**

Show (checked) or hide (unchecked) the Simulation Toolbar.

**Analysis Toolbar**

Not implemented in this version of the user-interface.

**Status Bar**

Show (checked) or hide (unchecked) the Status Bar. This bar contains: Command Prompts, System Status, Simulation Time, Clock Type, and the State Diagram Updating, Watch Updating states.
Refresh Views

Updates the Tree, State Hierarchy, and Watch Window. This is useful for updating the State Diagram and Watch Window when their dynamic updating have been disabled (via the Simulation Menu) and you wish to view the current state of the system.

Watch Window

Shows the Watch Window.

4.3.4 Simulation Menu and Toolbar

![The Simulation Menu and Toolbar](image)

**Figure 23** The Simulation Menu

**Figure 24** The Simulation Toolbar

**Run Atomic**

Runs one step in the active simulation.

**Run Microstep**

Runs one microstep in the active simulation.
Run Step

Runs one step in the active simulation.

Run Full Speed

Runs the active simulation at full speed.

Stop Execution

Stops the running simulation.

Stopping Conditions

Activates the Stopping Conditions dialog box wherein you may View, Add, or Remove Stopping Conditions.

Connect Channels

Activates the Channel Connector dialog box.

Force State Active

After the selection of a state in the State Diagram, you may force it to be Active here and also in a popup menu in the Diagram.

Force State Inactive

After the selection of a state in the State Diagram, you may force it to be Inactive here and also in a popup menu in the Diagram.

State Diagram Updating

Check or uncheck to either enable or disable the updating of the State Hierarchy Diagram during any run command executions.

Watch Window Updating

Check or uncheck to enable or disable the updating of the Watch Window during any run command executions.

Real Time

Set the simulation clock state to either Real Time (checked) or Simulation Time (unchecked).
4.3.5 State Hierarchy Menu and Toolbar

**Figure 25** The State Hierarchy Menu

**Figure 26** The State Hierarchy Toolbar

**Apply layout**

Apply the currently selected layout to the active specification. If you have made any modifications to a diagram and wish to apply the layout, then be certain to save your work via **Save Layout** if you wish to retain it; there is no reminder mechanism.

**Load layout**

You may load any layouts provided that they were created with this application's tool. The extension *af* is appended to your saved files to help identify them from within the file dialog boxes. If you load a layout that is inconsistent with the current specification, then **NIMBUS** will attempt to repair your graph. This feature can be used during incremental development of your system.
Save layout

Saves the current layout of the State Hierarchy Diagram. It is required that you save the diagram in a separate file from the rsml specification.

Copy to clipboard

Copies the Diagram layout to the Windows clipboard. This is useful for pasting the image into other applications such as word processors and image editors.

Save as wmf

Saves the layout as a Windows Metafile with a wmf extension. This is useful for saving the diagram in a portable format. WMF files can be inserted into numerous applications.

Select/View Layout

This option presents you with the Layout and Orientation Options page in the System Options dialog.

Select/View Colors

This option presents you with the Color Options page in the system options dialog.

Zoom Options

The zoom feature works as follows. The user selects a zoom factor. This factor is used for both zooming in and zooming out. The user can, at any time, reset to the original size by selecting None (see below).

Zoom Out (+)

Zoom out using the current factor.

Zoom In (-)

Zoom in using the current factor.

Set Zoom Factor: Percentages

You may select a zoom factor here. The current one is checked.

Set Zoom Factor: None

When checked, this option sets the diagram size to its original scale and then inhibits further zooming until this option is unchecked.

Set Zoom Factor: Fit to Window
The diagram is stretched to fit the currently visible viewing area of the State Hierarchy View.

**Highlight active states**

You may toggle the highlighting of active states. When this option is checked, active states are colored according to their fill color (a reconfigurable option from Color Options). Unchecking this item disables highlighting and is a useful feature for copying or saving structurally focused diagrams.

### 4.3.6 Tree View

Specification details are contained in the Tree View. The first level of this view displays identifying names for key components in the system. The tree is refreshed when: a new specification is loaded, when the user presses F5, and when the user selects Refresh Views from the View Menu.

At the top of the tree, the state hierarchy is displayed where represents a State and represents an Equivalence Class.

Variables are displayed with information for Type, Expecting Min, Expected Max, and Initial Values (input variables only).
Interfaces are displayed with their constant attributes as well as the configurable Connection information. This is refreshed with new information when the user collapses then expands the connection or interface name nodes, or by pressing F5 or selecting Refresh Views from the View Menu.
4.3.7 State Hierarchy Diagram

The State Hierarchy Diagram is a visualization of the hierarchy of states. The user may configure its Color Options, Layout and Orientation Options, and other settings. You are free to select and then move, resize, or stretch the states and their connective links. All relationships to the specification are preserved.

Active States are colored as shown, with atomic states filled and non-atomics bordered. In the following example, the state One is active. Also, the user has selected state Five and is about to Force State Active via the popup menu. You will notice that Force State Inactive as well as other diagram options is accessible from this menu as well.

Figure 27 State Hierarchy Popup Menu: Forcing a State Active

4.3.8 Status Bar

The Status Bar contains information about the current system state and settings as shown.

Figure 28 Status Bar
4.3.9 Stopping Conditions

![Stopping Conditions Viewer](image)

**Figure 29** The Stopping Conditions Viewer

To add a stopping condition, push the **Add** button.

![Add a Stopping Condition](image)

**Figure 30** Stopping Conditions Example: Variable Change

In the next dialog, select the radio button for the item to which you want to add a stopping condition. In this example the user chooses to add the condition *When Out_Var changes*:
For states, select the item in the tree diagram and then choose the desired condition from the list. Here, the user adds the condition "When state Three is entered".

The results are shown above. You may delete a condition: select it from the list and press the Delete button.
4.3.10 Watch Window

The Watch Window contains a listing of Variables and their values. When values change over time, the newly updated ones are colored red as shown.

![Watch Window](image)

**Figure 33** Watch Window

4.3.11 Channel Connector

To connect an interface to a channel:

- Select the interface from the list
- Choose the connection type **File** or **NIMBUS Channel**
- **Select the appropriate options**:
  - For files, choose Single or Multiple and enter the source file name.
  - For NIMBUS Channels that are Publish or Read, you have the option to select **Buffer**.

Here the user connects **In_IFace** to the file channel **basic_in.txt**:
4.4 Running a Simulation

Here are a few quick steps that enable you to get simulation started.

Load a RSML-e File
1. Load a rsml file from File menu or by clicking the icon to the left of the main toolbar
2. Connect Channels with the Channel Connector.
3. Add Stopping Conditions (optional)
4. Check/Set the Clock Type
5. Check/Set the Updating of the State Hierarchy Diagram and Watch Window
6. Bring the Watch Window to the foreground.
7. Run: Atomic, Microstep, Step, or Full Speed
8. Stop

Optionally, the user could have created a Script File of Command Line Interpreter directives and then loaded that file into the simulator.

Once a specification is loaded, you can visualize its structure with the Tree View and State Hierarchy Diagram.
As the execution proceeds, you can view its progress by observing: the highlighting of active states in the State Hierarchy Diagram, the changing of variable values in the Watch Window, the updating of the clock time in the Status Bar.
### 4.5 Command Reference

The following is a summary of commands that are accessible from the graphical user interface and the command line interpreter. There are some additional commands, not listed, that are unique to the GUI and the Command Line Interpreter. Use the help command to get the latest information on all commands available to the user.

<table>
<thead>
<tr>
<th>Task</th>
<th>Graphical User Interface</th>
<th>Command Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Help</td>
<td>[Help Menu</td>
<td>Main Toolbar]</td>
</tr>
<tr>
<td>Load a Specification</td>
<td>[File Menu</td>
<td>Main Toolbar</td>
</tr>
<tr>
<td>View Specification</td>
<td>Tree View &amp; State Hierarchy View</td>
<td>dump -s allStates</td>
</tr>
<tr>
<td>View Active States</td>
<td>State Hierarchy View (highlighted)</td>
<td>dump -s activeStates</td>
</tr>
<tr>
<td>View Input Interfaces</td>
<td>Tree View</td>
<td>dump -s inputHandlers</td>
</tr>
<tr>
<td>View Output Interfaces</td>
<td>Tree View</td>
<td>dump -s outputHandlers</td>
</tr>
<tr>
<td>View Input Variables</td>
<td>Tree View</td>
<td>dump -s inputVariables</td>
</tr>
<tr>
<td>View Output Variables</td>
<td>Tree View</td>
<td>dump -s outputVariables</td>
</tr>
<tr>
<td>Connect Channels</td>
<td>[Simulation Menu</td>
<td>Simulation Toolbar] Channel Connector</td>
</tr>
<tr>
<td>View Stopping Conditions</td>
<td>[Simulation Menu</td>
<td>Simulation Toolbar] Stopping Conditions</td>
</tr>
<tr>
<td>Add Stopping Conditions</td>
<td>[Simulation Menu</td>
<td>Simulation Toolbar] Stopping Conditions Add</td>
</tr>
<tr>
<td>Remove Stopping Conditions</td>
<td>[Simulation Menu</td>
<td>Simulation Toolbar] Stopping Conditions Remove</td>
</tr>
<tr>
<td>Run One Atomic Step</td>
<td>[Simulation Menu</td>
<td>Simulation Toolbar</td>
</tr>
<tr>
<td>Run One Microstep</td>
<td>[Simulation Menu</td>
<td>Simulation Toolbar</td>
</tr>
<tr>
<td>Run One Step</td>
<td>[Simulation Menu</td>
<td>Simulation Toolbar</td>
</tr>
<tr>
<td>Run Full Speed</td>
<td>[Simulation Menu</td>
<td>Simulation Toolbar</td>
</tr>
<tr>
<td>Stop</td>
<td>[Simulation Menu</td>
<td>Simulation Toolbar</td>
</tr>
<tr>
<td>Force State Active</td>
<td>[Simulation Menu</td>
<td>Simulation Toolbar]</td>
</tr>
<tr>
<td>Force State Inactive</td>
<td>[Simulation Menu</td>
<td>Simulation Toolbar]</td>
</tr>
<tr>
<td>Set Time Model to Real-time</td>
<td>Simulation Menu: ✔ Realtime</td>
<td>setoption driver realtime</td>
</tr>
<tr>
<td>Set Time Model to Sim-time</td>
<td>Simulation Menu: Realtime</td>
<td>setoption driver simtime</td>
</tr>
<tr>
<td>Load a Script File</td>
<td>[File Menu</td>
<td>Simulation Toolbar]</td>
</tr>
<tr>
<td>State Hierarchy Layout</td>
<td>[State Hierarchy Menu</td>
<td>Toolbar]</td>
</tr>
<tr>
<td>Options and Color Options</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>-----------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>State Hierarchy Copy, Load, Save, Zoom, Apply, Layout</td>
<td>[State Hierarchy Menu</td>
<td>Toolbar]</td>
</tr>
<tr>
<td>State Hierarchy Active State Highlighting</td>
<td>State Hierarchy Menu: ✔Highlight Active State</td>
<td></td>
</tr>
<tr>
<td>Completeness and Consistency Analysis</td>
<td>ceAnalysis [subcommand]</td>
<td></td>
</tr>
<tr>
<td>Backwards Execution Analysis</td>
<td>beAnalysis [subcommand]</td>
<td></td>
</tr>
</tbody>
</table>
5 Using NIMBUSChannel

5.1 Introduction

Specification and verification of software for safety critical systems is a difficult problem. There are three methods available to ensure the correctness of such systems: manual inspections, formal verification, and simulation and testing. All three approaches have various strengths and weaknesses; however, they are complementary. To achieve the high level of confidence in the correctness of the system necessary today's applications, all three approaches must be used in concert. The NIMBUSChannel communication framework supplies an important component of the simulation and testing aspect of the NIMBUS environment.

The NIMBUSChannel framework was developed to support the execution of RSML\textsuperscript{-e} models in conjunction with models of the other components in the environment. The framework was conceived in the work done at the University of Minnesota in the Critical Systems Research Group. During their work with RSML, the group developed a simulation that was capable of executing the formal definition of the language while taking input from text files. Nevertheless, the group discovered that these capabilities alone were not a sufficiently powerful environment in which to test a formal specification. It was prohibitively inflexible to create the text file input scripts in the early stages of execution because, most of the time, the user did not know exact sequences of input to test; they only wished to "debug" the formal specification. Furthermore, in cases where a dynamic model, for example, an aircraft, was involved, it was very difficult to create even semi-realistic input scripts for the simulation. These experiences led to the creation of the following requirements that any environment for the debugging and testing of formal specifications should support.

- It must support execution of the RSML\textsuperscript{-e} model while interacting with accurate models of the component's environment. These models should be able to be nearly anything the analyst might desire to model the component be that other RSML\textsuperscript{-e} specifications, software simulations, or even actual hardware in the environment.

- It must allow the analyst to easily modify and interchange the models of the components in the environment. Preferably, the components should be dynamically swappable at run time.

- As the specification is refined from high level requirements there should not be any large conceptual leaps in the way in which the control software communicates with the environment.

To meet these requirements, it was clear that some standard inter-process communication method would be necessary. Furthermore, the more high-level such a mechanism was, the better it would support the needs of NIMBUSChannel. Therefore, the CriSys group examined two technologies: COM (Component Object Model) and CORBA (Common Object Request Broker Architecture). These frameworks provide a high-level object-oriented communication interface, and allow transparent distributed object access. In the end, COM was selected because it was freely available for the target platform (Windows NT) and also supported by a greater number of
current applications and scripting technologies (i.e. Visual Basic for Applications and ActiveX scripting); thus, allowing greater flexibility at decreased cost.

The framework models interprocess communication as messages passing over simple channels. Currently, the only types supported by the channels are integer, enumerated, and boolean; in the future, the channels will most likely support floating point as well. Formally, all communication from a RSML\textsuperscript{e} specification is one of two types: Send-Receive or Publish-Read. Send-Receive communication is a message passing protocol where the source pushes a message across the channel to the destination, causing an interrupt, or signal, in the receiving process. This type of communication is well suited to COM's function-call type of interaction. Publish-Read communication is where the source updates (or Publishes) the message on the channel intermittently and at any time the destination can pull the current message from the channel. This type of communication requires additional support because there must be a buffer (to hold the current message on the channel). There are a number of places that make logical sense for this buffer to reside.

- **Inside the publishing application.** In this model, when the destination of the channel wishes to read, the source of the channel is interrupted and produces the message requested. Storing the buffer here allows, for example, the destination of the channel to cause an update of a spreadsheet. Notice that in this model the publishing end of the channel never calls the "Publish" method of the channel-end component; instead, it simply waits to be called by the destination end of the channel. The subtype of the Publish-Read communication is known as PublishViaEvent-Read.

- **Inside the publishing channel component.** Alternatively, the buffer could reside inside the publishing end of the channel. Then, instead of passing it immediate over the channel, the channel-end component stores the message and waits to be called by the reader. When the reader asks for the message on the channel, the publishing channel-end simply passes the buffered message across the channel. This type is known as PublishWithBuffer-Read.

- **Inside the reading channel component.** Finally, the publishing end can immediately pass the message over the channel (like a Send-Receive channel) and the read channel, instead of interrupting the destination like in Send-Receive, stores the message in its buffer. Then, when the reading process need the message, the reader channel-end simply returns its buffered copy. This type is known as Publish-ReadWithBuffer.

These different options allow the user of NIMBUSChannel to optimize the number of inter-process COM calls and thereby increase the effective speed of the simulation environment. Furthermore, the ability to publish by event (i.e. interrupt the publishing process) allows for great flexibility in integrating applications like spreadsheets and databases into the NIMBUSChannel environment. More information about how these types of communication are implemented can be found in the Behind the Scenes section on page 56.

This document describes the NIMBUSChannel: its component applications, how to manage a simulation, how to create and add component models in the environment. To conserve space, this document assumes some familiarity with COM, C++, and Visual Basic. The examples given should be readable by any computer scientist; however, the reader should not expect to actually
be able to construct a NIMBUSChannel component without becoming familiar with the
technologies involved.

5.2 Components of NIMBUSChannel

Aside from the NIMBUSChannel clients provided in the example applications, the NIMBUS
Environment installation includes three main NIMBUSChannel client applications: The
NIMBUSim RSML-e Simulator and Analysis tool, the NIMBUSChannel MFC client, and the
NIMBUS Manager. These applications provide a basis from which to build a system simulation.
A dynamic link library (DLL), type libraries, and a background executable process support them.
This section describes each of these components and their use.

5.2.1 NIMBUSim

NIMBUSim is described in *The NIMBUSim Graphical User Interface Manual* included in the
NIMBUS environment documentation. To connect the interface of a RSML-e specification to the
NIMBUSChannel framework, simply select the interface you wish to connect in the channel
connection dialog box and then choose NIMBUSChannel as the type of connection. For Publish
or Read interfaces it is possible to select whether or not the buffer resides with the selected
interface, or on the other end of the channel (see page 56, Behind the Scenes).

Figure 35 shows the channel connection dialog box. The example ASW specification has been
loaded and the user is in the process of connecting the DOIStatusMessageInterface to the
NIMBUSChannel environment. Because the DOIStatusMessageInterface specifies the receive
type of communication, the Buffer option is not available and is grayed out.

![Figure 35: The channel connector dialog box in NimbusSim with COM selected](image)
5.2.2 **NIMBUSChannel Client**

The NIMBUSChannel Client allows the user to send simple on-the-fly inputs to other components in the NIMBUSChannel architecture. Figure 36 shows the configuration dialog box for the client. The user specifies the Component name, the channel name, the type of communication, and the number of fields in the message that they are sending. The channel name is a string that identifies which channel in the NIMBUSChannel environment the client is connecting to. Channel names in the NIMBUSSim client are given in the specification document. The component name identifies the process (or component) supplying that end of the channel.

![Figure 36: The configuration dialog for the simple MFC client](image)

Once the user has selected the options necessary for channel configuration and pressed the OK button, the client creates a new channel-end object of the appropriate type and registers itself in the NIMBUSChannel framework. At that point, it is ready to participate in the framework and the main window of the client application appears.

![Figure 37: The simple MFC client connected to the AltitudeChannel](image)

Figure 37 shows the main window of the MFC client application. The window supports all the various types of communication. The title bar shows the channel name on the left and the component name on the right so that the client's place in the framework can be easily identified.

5.2.3 **NIMBUS Manager**

The NIMBUS Manager allows the user to dynamically control the connections between the various components which are registered with the NIMBUSChannel system. Figure 38 shows the main window of the NIMBUS Manager. In the left-hand column, all the channels currently
registered in the system are listed. When the user clicks on a channel name, the detailed information about that channel is displayed in the area to the right of the channel list.

For each channel, the NIMBUS Manager displays the channel name (in Figure 38, AltitudeChannel), the type (Send-Receive) and three lists (sources, destinations, and observers). In the lists are the components that supply that channel. For example, Figure 38 shows that Altitude channel currently has two sources and two destinations. The sources are "MS Excel ASW System" from the example files and "Simple MFC Client app" which was instantiated in the previous section.

![Image: The main window of the NIMBUS Manager](image)

**Figure 38:** The main window of the NIMBUS Manager

The check marks beside the component names in Figure 38 indicate whether or not that component is currently active on the channel or not. A component that is not active cannot participate in the message processing (sending, receiving, publishing, etc) on that channel. Thus, Figure 38 shows that the "MS Excel ASW System" component is active, whereas the "Simple MFC Client app" is not. Notice in the figure that both "NIMBUSim (ASW)" and "VB Pilot's Display" are active destinations on the AltitudeChannel. The NIMBUSChannel environment allows multicast communication to allow multiple different displays of the data. Also (not pictured in Figure 38) allowed are observers on a channel.

The NIMBUSChannel environment *does not* allow multiple sources on a channel. Thus, if the user were to click on the checkbox next to "Simple MFC Client app," then it would become the active source on the AltitudeChannel and "MS Excel ASW System" would become inactive. For convenience, the first registered source and destination on a channel are made active by default.
5.2.4 Behind the Scenes

The capabilities of NIMBUSChannel are implemented by a number of COM objects that implement several custom interfaces. This section explains briefly what a COM object is, what COM objects are used in NIMBUSChannel, and how the NIMBUSChannel objects are stored and implemented.

COM is an executable (binary) software component standard. Like Java, each COM object can support multiple interfaces. Each interface has a number of methods that can be called by the user of the object. Furthermore, to support dynamic run-time discovery of a component's capabilities, every COM interface inherits from an interface called IUnknown. The purpose of IUnknown is to allow the user to query for another interface on the object. If the query succeeds, then the client has a pointer to the requested interface. If not, the client receives an error message indicating that the interface as not supported.

COM is a binary standard. That is, as long as an object conforms to the binary specification it can be loaded into memory and run by the COM library. Therefore, COM is language independent. Currently, developers can create COM objects in C++, Visual Basic, or Java under a number of different development environments. The procedure for creating a COM object is to 1) write the code and compile it, and 2) register it with the COM library on the machine you wish it to run on. It is also possible to transparently call objects on different machines using DCOM.

Under Windows, COM objects are either packaged into an executable file, which runs on its own, or into a dynamic link library, which is loaded into a process. The NIMBUSChannel environment uses both techniques to package the various objects in the system. There are three types of objects in NIMBUSChannel:

- **Channel Ends** are the objects that implement the details of the communication channels (i.e., implement Send-Receive, Publish-Read as discussed in the Introduction). There are a total of nine channel ends: Send, Receive, Publish, PublishWithBuffer, PublishWithEvent, Read, ReadWithBuffer, and, Observe. These objects shield the client processes from the details of the specific type of communication. They also handle all communication with the manager application.

- **Channel Wrapper** objects provide a convenient method of creating and destroying the channel end objects in the target development environment. Currently, there are channel wrapper objects available for both C++ and Visual Basic.

- The **Channel Manager** object manages the connection of channels in the system. This is a background process that is separate from the aforementioned NIMBUS Manager. The channel manager houses all the data structures displayed by the NIMBUS Manager and is the object which actually does the connecting and disconnecting of channels.

---

1 Many references for COM are available. Among the best is Inside COM by Dale Rogerson (Microsoft Press).
The channel ends and channel wrapper objects are located in the RSMLChannelEnds.dll file whereas the channel manager is located in the RSMLChannelManager.exe file. Both these files are located in the distribution directory and are registered in the Windows registry.

5.3 Connecting Other Applications

Connecting other applications into the NimbusChannel framework boils down to getting those applications to create and use the channel wrapper and/or channel end objects discussed at the end of the previous section. There are several important questions to ask about the application that you wish to integrate:

- Does the application have an API or macro capability that would allow other applications access to its internal data structures? If not, do you have access to the source code of the application?
- Does the application support ActiveX scripting technologies, for example, Microsoft's Visual Basic for Applications.
- Does the application support OLE/COM automation.

The ease of integrating any application into NimbusChannel is dependent upon how full featured the API of that application is. Of course, if the application does not provide any API and there is not access to the source of the application, it will be impossible to easily integrate it into the NimbusChannel framework. If there is an API accessible through, for example, C or C++ then a NimbusChannel C++ Client might be the way to go unless a better alternative is available.

If the application supports OLE automation, then the application supports a set of COM interfaces that allow Visual Basic (or any other application) to cause various commands of the application to be run. Essentially, it allows for remote control of the application. Thus, the control the application and the channels can be written in one Visual Basic application and the developer can use the VB wrapper.

The easiest and fastest way to achieve integration is to use an application that supports ActiveX scripting. ActiveX scripting is a standard where applications support the creating and use of COM objects within the context of a built-in scripting or macro language. Visual Basic for Applications (VBA) is one common example of an ActiveX scripting language. Using a language like VBA is absolutely the fastest way to integrate a new application into the system and it allows the most flexibility.

The general procedure that is followed in creating clients for NimbusChannel is the following.

1. Declare a variable to hold the instance of the channel wrapper object.
2. Create the channel wrapper object.
3. Configure the channel wrapper object by filling in the channel name, component name, and channel type properties.
4. Call the register method to have your new channel join the NimbusChannel environment.
5. Use the channel.
6. Call the unregister method when you are through using the channel.

The following sections describe the creating of a visual basic and a C++ Client. The focus is on the code which is necessary to make the application work in the NimbusChannel framework, not on the functionality of the application. Thus, much code that deals with, for example, the user interface of the application is omitted.

### 5.3.1 Building a Visual Basic Client

Visual basic clients (including VBA clients) are the easiest to make. The example in this section shows the creation of a client that has two channels: sending and receiving. The example omits any of the user interface code normally associated with a VB application and instead focuses on the code necessary to make the channels function.

Below, you see the first step in creating a NimbusChannel client: the variables for the objects have been declared.

```
Dim WithEvents MySendChannel As RSMLChannelVBWrapper
Dim WithEvents MyReceiveChannel As RSMLChannelVBWrapper
```

These variables are declared "WithEvents" because for the receive, observe, and publish via event channel types, the channel end objects will cause an event to occur in Visual Basic. Note, that the caller (i.e., the sender) is blocked until the event procedure invoked by the receive event is finished executing. Therefore, it is highly undesirable to make additional NimbusChannel calls within the event procedures as this would most likely cause a deadlock.

Object variables in Visual Basic are not initialized automatically to new objects. Thus, somewhere in the code, it is necessary to set the MySendChannel variable to a new instance of the RSMLChannelVBWrapper object. After the object is created, the programmer fills in the properties necessary to initialize the object: the channel name, component name, and channel type. Finally, the object is registered with the channel manager by calling the "Register" method. This code is shown on the following page.

```
Sub InitChannels()
    ' MySendChannel
    Set MySendChannel = new RSMLChannelVBWrapper
    With MySendChannel
        .ChannelName = "SendingChannel"
        .ChannelType = rsmlSend
        .ComponentName = "My First VB Client"
    End With
    MySendChannel.Register
    ' MyReceiveChannel
```
Set MyReceiveChannel = new RSMLChannelVBWrapper

With MyReceiveChannel
    .ChannelName = "ReceivingChannel"
    .ChannelType = rsmlReceive
    .ComponentName = "My First VB Client"
End With

MyReceiveChannel.Register

End Sub

After the InitChannels function (above) has completed, both the sending and receiving channel clients will be ready for use. For the sending channel, we can make a VB interface form to allow the user to send the message (not shown). However, for the receiving VB will call the event procedure pictured below.

Sub MySendChannel_receive(msg() As Long)
    ' Display Message shows the message to the user
    DisplayMessage msg
End Sub

The procedure takes one argument: an array of long integers. This array is the message that is passed over the channel. Enumerated types and Boolean values are converted to integers (0 for false, 1 for true, 0 indexed for the enumerated types) before they are passed over the channel. The developer can put any code that he or she likes in the event procedure; however, note again that the sender is blocked until the call returns.

Sub CleanupChannels()
    MySendChannel.Unregister
    MyReceiveChannel.Unregister
End Sub

Finally, after the application is done using the NimbusChannel framework it should unregister the channel end objects so that the channel manager can disconnect and remove them from the system.

5.3.2 Building a C++ Client

Creating a C++ Client is much like creating a Visual Basic client. The differences many lie in syntax and the fact that in C++ it is necessary to deal with VB compliant types, for example, the SAFEARRAY structure that is used to pass the messages over the channels. These topics are beyond the scope of this guide, and, in practice, creating a custom C++ client is a rare occurrence under Windows. An example of a C++ Client, written as a dialog based MFC application, is the simple channel client distributed with the Nimbus environment. The source for this client can be obtained from Jeff Thompson (thompson@cs.umn.edu).
5.4 Troubleshooting and Common Issues

This section is under development and will evolve as the tools get more use.
6 References


Appendix A - Using the Airlock Interface

The VB Airlock interface is used to provide the environment for an airlock control specification. It uses the NIMBUS-Channel communications framework to communicate with a specification. The VB program periodically sends a message to the specification describing the status of the environment and receives a message with actuator commands from the specification. The input message for the specification is sent over the channel named “inputChannel”, and the VB client receives the output message from the specification over the channel named “outputChannel”. By connecting an RSML specification to these channels, it is possible to communicate with the VB client.

A.1 Message Specifications

The message specifications are as follows:

MESSAGE Environment_Message {
    f_panic_button IS boolean,
    f_reset_button IS boolean,
    f_inner_door_request IS boolean,
    f_outer_door_request IS boolean,
    f_inner_door_open IS boolean,
    f_outer_door_open IS boolean,
    f_airlock_occupied IS boolean
}

This message is received by the specification, and describes the status of, respectively:

- the panic alarm button
- the reset button
- the inner door request button
- the outer door request button
- whether or not the inner door is open
- whether or not the outer door is open
- whether or not the airlock is occupied
From this information, it should be possible to create a specification controlling the behavior of the airlock.

The message sent to control the actuators of the airlock is as follows:

```
MESSAGE Actuator_Message {
    f_inner_door_lock IS boolean,
    f_outer_door_lock IS boolean,
    f_decontaminate IS boolean,
    f_panic_alarm IS boolean,
    f_timeout_alarm IS boolean
}
```

This message describes actuator commands to, respectively:

- lock (TRUE) or unlock (FALSE) the inner door
- lock (TRUE) or unlock (FALSE) the outer door
- decontaminate the chamber
- turn on/off the panic alarm
- turn on/off the timeout alarm.

These actuators control the operation of the airlock.

### A.2 Creating Specification Interfaces.

In order to connect the specification to the VB Airlock, it is first necessary to create interfaces to receive/send messages. First, we need an input interface to receive in the environment message. Here is some sample code that will do this step:

```
MESSAGE Environment_Message {
    f_panic_button IS boolean,
    f_reset_button IS boolean,
    f_inner_door_request IS boolean,
    f_outer_door_request IS boolean,
    f_inner_door_open IS boolean,
    f_outer_door_open IS boolean,
    f_airlock_occupied IS boolean
}
```

```
IN_VARIABLE panic_button : boolean
    INITIAL_VALUE : FALSE
    CLASSIFICATION: MONITORED
END IN_VARIABLE

IN_VARIABLE reset_button : boolean
    INITIAL_VALUE : FALSE
    CLASSIFICATION: MONITORED
END IN_VARIABLE
```
IN_VARIABLE inner_door_request : boolean
   INITIAL_VALUE : FALSE
   CLASSIFICATION: MONITORED
END IN_VARIABLE

IN_VARIABLE outer_door_request : boolean
   INITIAL_VALUE : FALSE
   CLASSIFICATION: MONITORED
END IN_VARIABLE

IN_VARIABLE inner_door_open : boolean
   INITIAL_VALUE : FALSE
   CLASSIFICATION: MONITORED
END IN_VARIABLE

IN_VARIABLE outer_door_open : boolean
   INITIAL_VALUE : FALSE
   CLASSIFICATION: MONITORED
END IN_VARIABLE

IN_VARIABLE airlock_occupied : boolean
   INITIAL_VALUE : FALSE
   CLASSIFICATION: MONITORED
END IN_VARIABLE

IN_INTERFACE Update_Interface :
   MIN_SEP : UNDEFINED
   MAX_SEP : UNDEFINED
   INPUT_ACTION : RECEIVE(Environment_Message)
   HANDLER :
      CONDITION : TRUE
      ASSIGNMENT
         panic_button := f_panic_button,
         reset_button := f_reset_button,
         inner_door_request := f_inner_door_request,
         outer_door_request := f_outer_door_request,
         inner_door_open := f_inner_door_open,
         outer_door_open := f_outer_door_open,
         airlock_occupied := f_airlock_occupied
      END ASSIGNMENT
   END HANDLER
END IN_INTERFACE

Similarly, the output interface must be connected:

MESSAGE Actuator_Message 
   { 
      f_inner_door_lock IS door_lock_status,
      f_outer_door_lock IS door_lock_status,
      f_decontaminate IS boolean,
      f_panic_alarm IS on_off,
      f_timeout_alarm IS on_off
   }
OUT_INTERFACE Actuator_Interface:
    MIN_SEP : UNDEFINED
    MAX_SEP : UNDEFINED
    OUTPUT_ACTION : SEND(Actuator_Message)
    HANDLER :
        CONDITION : TRUE
        ASSIGNMENT
            f_inner_door_lock := inner_door_lock,
            f_outer_door_lock := outer_door_lock,
            f_decontaminate := decontaminate,
            f_panic_alarm := panic_alarm,
            f_timeout_alarm := timeout_alarm
        END ASSIGNMENT
    ACTION : SEND
    END HANDLER
END OUT_INTERFACE

where inner_door_lock, etc., should be state variables within your specification.

A.3 Connecting the RSML Channels

In order to “wire together” the VB client with the RSML specification, you must connect the RSML interfaces to the VB interfaces. On the VB side, this is embedded in the application; however, we must manually connect the interfaces on the RSML side. You can either connect the interfaces from the RSML command line, or you can write a script file, which will connect the interfaces whenever the specification loads, which is the recommended procedure.

Creating a Script File

To create an automatically loading script file, create a text file with the name <SPECIFICATION_NAME>.nscript, where SPECIFICATION_NAME is the name of the RSML specification. If this file is in the same directory as the specification, it will be automatically loaded when the specification is loaded.

Script File Contents

Here is an example script file that connects the Update_Interface and the Actuator_Interface (described above) to the channels used by the Visual Basic client. The name of the RSML specification that uses this file is Cleanroom.nimbus.

Cleanroom.nscript:
    connect -i Update_Interface inputChannel -com
    connect -o Actuator_Interface outputChannel -com
    setoption driver realtime

The connect command connects an interface to a communications channel. The –i option is used to specify that this connection should be for input into the specification. Similarly, –o signifies output from the specification. Next are the names of the interface and channel to be connected,
respectively. The final argument specifies that this channel is a COM channel. Nimbus can read/write over several channel types: for example, it is possible to create input files and use them as channels. COM channels use Microsoft’s COM to communicate in real time between applications.

The `setoption` command is used to toggle several simulation options. In this case, it sets the simulator to run in `realtime` mode, as opposed to `simulated time` mode. In `simulated time` mode, the specification runs much faster than real time. This mode is useful when replaying a log of generated events or using file channels. However, for COM channels, the specification must be run in real time mode for correct results.

### A.4 Executing the Nimbus Simulator and the VB Driver

Once a script file has been created, it is straightforward to simulate the behavior of the specification. First, start the VB Driver executable. Then, start the RSML simulator, and load the specification. At this point, if your script file is correctly written, all channels should be wired together. You can check by looking at the Nimbus command line window after loading the specification; it should read something like:

```
Script file name: C:\data\srcsafe\Members\whalen\csci8990\cleanroom.nscript
executing command: connect -i Update_Interface inputChannel -com
Finding channel manager.
executing command: connect -o Actuator_Interface outputChannel -com
Finding channel manager.
executing command: setoption driver realtime
executing command:
completed script file C:\data\srcsafe\Members\whalen\csci8990\cleanroom.nscript
5 lines processed.
```

If you don’t see these lines in the command line window, it is likely that the file name of the script file is wrong; make sure that the file does not have a .txt extension on the end.

Finally, you should be able to simply start the simulator and test your work.
The VB Interface looks like this:

When the checkboxes are checked, the “actions” are occurring. So by checking “Open Inner Door” (if the door is unlocked), the user has opened the inner door. These conditions continue to hold until the box is unchecked.

The buttons at the bottom (reset and PANIC), simulate the user pressing either the reset or the panic button. In this implementation, these activities are only true for one instant and then false thereafter.

The status lights determine the state of the system. Red and Green mean locked and unlocked, respectively, for the doors. Similarly, Green means no alarm, and red means alarm on.

That’s about all there is to it. Have fun & let us know of any errors.
Appendix B - Textual Grammar of RSML\textsuperscript{$e$}

The full grammar for RSML\textsuperscript{$e$} is given below.

\begin{verbatim}
component_def : def_list ;
def_list : /* empty */ | def_list def ;
def : type_def | constant_def | state_variable_def | in_variable_def | in_interface_def | out_interface_def | macro_def | function_def | message_def ;

/*----------- State definition rules -------------------------------*/
state_variable_def : STATE_VARIABLE IDENTIFIER array_decl ':'
variable_type_decl
    PARENT ':' parent_decl
    INITIAL_VALUE ':' expression /* checked to be const */
variable_numeric_decl
    variable_numeric_decl
    classification_def
    case_list
    END STATE_VARIABLE ;

variable_numeric_decl : /* empty */ | UNITS ':' IDENTIFIER /* added to variable properties */
properties */
    EXPECTED_MIN ':' expression /* checked to be const */
const */
    EXPECTED_MAX ':' expression /* checked to be const */
const */
variable_type_decl : type_ref | VALUES ':' '{' enum_element_list '}'
\end{verbatim}
array_decl : /* empty */
    | ['[ expression TO expression ']' /* both expressions checked to be const */

parent_decl : NONE
    | parent_name_path

parent_name_path : IDENTIFIER
    | parent_name_path '.' IDENTIFIER

classification_def : /* empty */
    | CLASSIFICATION ':=' IDENTIFIER

/*---------------- Type definitions, Only for enumerated types ----------*/

/* The type definitions can be safely ignored, as they should have been handled in the first pass. */
type_def : TYPE_DEF IDENTIFIER '{' enum_element_list '}'

enum_element_list : IDENTIFIER
    | enum_element_list ',' IDENTIFIER

type_ref : IDENTIFIER
    | INTEGER_TYPE
    | REAL_TYPE
    | BOOLEAN_TYPE
    | TIME

/*---------------- Message definition rules --------------------------*/

message_def : MESSAGE IDENTIFIER '{' field_list '}

field_list : /* empty */
    | IDENTIFIER IS type_ref
    | field_list ',' IDENTIFIER IS type_ref

/*---------------- Constant definition rules ------------------------*/

constant_def : CONSTANT IDENTIFIER ':=' type_ref
    | UNITS ':=' IDENTIFIER
    | VALUE ':=' expression /* checked to be const */
    END CONSTANT
CONSTANT IDENTIFIER ':' type_ref  
    VALUE ':' expression /* checked to be const */ 
END CONSTANT

/*------------ Variable definition rules ----------------------------*/

in_variable_def  : IN_VARIABLE IDENTIFIER 
    array_decl ':' type_ref 
    INITIAL_VALUE ':' expression /* checked to be const */
const */
    variable_numeric_decl 
    classification_def 
END IN_VARIABLE

/*------------- Input Interface definitions --------------------------------*/

in_interface_def : IN_INTERFACE IDENTIFIER ':' 
    MIN_SEP ':' expression /* checked to be const */
const */
    MAX_SEP ':' expression /* checked to be const */
const */
    INPUT_ACTION ':' in_interface_type_spec '(' 
IDENTIFIER ')' 
    in_handler_list 
END IN_INTERFACE

in_interface_type_spec : RECEIVE 
| READ

in_handler_list : in_handler 
| in_handler in_handler_list

in_handler : in_handler_type ':' 
    CONDITION ':' condition 
    in_assignment 
END HANDLER

in_handler_type : RECEIVE_HANDLER 
| HANDLER

in_assignment : /* empty */ 
| ASSIGNMENT 
    in_assignment_list 
END ASSIGNMENT

Introduction to RSML and Nimbus
in_assignment_list : identifier_name_path ASSIGN_TOKEN expression
| in_assignment_list ',' identifier_name_path

out_interface_type_spec : SEND
| PUBLISH

output_handler_list : output_handler
| output_handler_list output_handler

out_handler_type : SEND
| PUBLISH
| NONE

optional_formal_parms : /* EMPTY */
| '(' formal_parameter_list ')'

Introduction to RSML® and Nimbus
macro_def : MACRO IDENTIFIER optional_formalParms ':
  condition
END MACRO

/*---------------- Function definitions -----------------------------*/
optional_expr_list : /* empty */
| expression_list
function_def : FUNCTION IDENTIFIER '(' formalParameter_list ')'
  case_list
END FUNCTION
| STUB_FUNCTION IDENTIFIER '(' formalParameter_list
')' ':
  optional_expr_list
END STUB_FUNCTION
;
case_list : /* EMPTY */
| case_list case
;
case : EQUALS expression IF condition
| TRANSITION expression TO expression IF condition
;
actual_parameter_list : /* empty */
| expression_list
;
/* formal parameter lists are set up in the first pass */
formal_parameter_list : /* empty */
| IDENTIFIER IS type_ref
| formal_parameter_list ',' IDENTIFIER IS type_ref
;

/*-------- Identifier path definition rules --------------------------*/

/
for the identifier_name_path rule, we don't need to check whether
an array_ref is NULL, because the rule will work correctly in either
case. Specifiers can leave array_ref blank in the case of a non-array
variable or if they wish to use an implicit 'this' expression.

We also assume that IDENTIFIER will always return non-NULL values.
*/
identifier_name_path : IDENTIFIER array_ref
| '.' '.' IDENTIFIER array_ref
array_ref  : /* empty */
           | '[' expression ']' ;
           ;

/*---------- Definitions defining AND/OR tables ----------------------*/
condition  : TABLE
            row_list
            END TABLE
            | expression /* Must return BOOLEAN */
            ;
row_list   : expression ':' truth_value_list ';
            | row_list expression ':' truth_value_list ';
            ;
truth_value_list : truth_value
               | truth_value truth_value_list
           ;
truth_value : 'T'
            | 'F'
            | '.'
            | '*'

/*---------- Expression definition rules ---------*/
expression : unary_expression
            | binary_expression
            | array_expression
            | event_expression
            | time_expression
            | prev_expression
            | '(' expression ')' |
            | ASSIGNED '(' identifier_name_path ')'
            | CHANGED '(' identifier_name_path ')'
            | IDENTIFIER '(' actual_parameter_list ')' |
            | literal
            | expression EQ_ONE_OF '{' expression_list '}'
unary_expression : '-' expression %prec UMINUS
                   | NOT expression
binary_expression : expression '*' expression
                   | expression '/' expression
                   | expression '+' expression
                   | expression '-' expression
                   | expression '>' expression
                   |
| expression '<' expression  
| expression LESS_OR_EQUAL expression  
| expression GREATER_OR_EQUAL expression  
| expression EQUAL expression  
| expression NOT_EQUAL expression 

array_expression : EXISTS '(' IDENTIFIER ',' 
| FORALL '(' IDENTIFIER ',' 
| identifier_name_path ',' expression ')' 

identifier_name_path , expression ')' 
| COUNT '(' IDENTIFIER ',' 
| FIRST_INDEX '(' IDENTIFIER ',' 
| identifier_name_path , expression ')' 
| LAST_INDEX '(' IDENTIFIER ',' 
| identifier_name_path , expression ')' 


event_expression : AT_TRUE '(' expression ')' 
| AT_FALSE '(' expression ')' 
| AT_CHANGED '(' expression ')' 

prev_step_expression : identifier_expression 
| PREV_STEP '(' identifier_expression ')' 

optional_pv : /* EMPTY */ 
| ',' INT_VALUE 

optional_ta : /* EMPTY */ 
| ',' INT_VALUE 

prev_expression : prev_step_expression 
| PREV_ASSIGN '(' prev_step_expression optional_pv ')' 
| PREV_VALUE '(' prev_step_expression optional_pv ')' 
| TIME_ASSIGNED 
| TIME_CHANGED 

static_variable_info_decl : EXPECTED_MIN 
| EXPECTED_MAX 
| UPPER_BOUND
| LOWER_BOUND |

```
identifier_expression : identifier_name_path |
                      identifier_name_path DOUBLE_COLON

static_variable_info_decl |
                          identifier_name_path DOUBLE_COLON MAX_SEP |
                          identifier_name_path DOUBLE_COLON MIN_SEP |
                          identifier_name_path DOUBLE_COLON THIS |
                          identifier_name_path DOUBLE_COLON TIME |
                          identifier_name_path DOUBLE_COLON LAST_IO

time_expression : TIME |
                  HOURS '(' expression ')' |
                  MINUTES '(' expression ')' |
                  SECONDS '(' expression ')' |
                  MILLISECONDS '(' expression ')'

expression_list : expression |
                 expression_list ',' expression

literal : time_literal |
         INT_VALUE |
         REAL_VALUE |
         TRUE_TOKEN |
         FALSE_TOKEN |
         UNDEFINED

time_literal : tl_comp_list

tl_comp_list : INT_VALUE tl_units |
             tl_comp_list tl_separator INT_VALUE tl_units

tl_separator : /* empty */ |
             AND

tl_units : H_TOKEN |
          MIN_TOKEN |
          S_TOKEN |
          MS_TOKEN
```