1/

NASA Technical Memorandum 107596

User's Guide to the Reliability Estimation System Testbed (REST)

David M. Nicol
Daniel L. Palumbo
Adam Rifkin

June 1992

NASA
National Aeronautics and Space Administration

Langley Research Center
Hampton, VA 23665

Contents

List of Figures

Chapter 1

Introduction to REST

1.1 Introduction

The Reliability Estimation System Testbed (REST) is a software system designed to support the hardware reliability analysis of complex fault-tolerant computer systems. REST is presently comprised of a system modeling language and translator, an X-Windows front-end to support the design of systems using the language, a model debugger, and model execution engines implemented both on serial workstations, and on the Intel iPSC multiprocessor. REST is used by first defining a system of interest in the description language. The model can then be fully analyzed using either the serial or parallel execution engines. REST output includes upper and lower bounds on the probability of encountering a failure condition, and summaries of the conditions under which those failures occur.

This documentation contains three basic parts. The first is a description of RML, the language used to model systems in REST. The second is a detailed example of modeling in RML, with comparisons with equivalent models in ASSIST[1]. The third is a description of the X-Windows based user interface to REST.

Chapter 2

The REST Modeling Language (RML)

2.1 Overview of RML

In this chapter we describe the language, RML, used by the REST to model a system. RML centers around the notion of a "module". A *module type* is an abstraction able to describe a range of entities, from simple hardware components up to complex organizations of such components. A module type definition in RML plays much the same role as a class definition in an object oriented language, or a struct definition in C—it serves to describe the properties of an entity that may be replicated many times throughout the system. The first step in defining a system in RML is to describe all the module types to be found in the system. Such declarations include state variables, rules for detecting a system failure condition, rules for modifying the system state, and rules for pruning system state modification. *Module variables*, i.e., instances of module types, are then declared. Next, the interconnection paths between module variables are described, and finally, the initial states of the module variables are defined. The model analysis portion of REST takes the system so described and repeatedly transforms the system state in accordance with rules given with the module type definition.

The REST translator maps all state variables implicit in the declaration of module variables onto a global state vector. However, an RML expression of a module and its behavior is done in terms of state variables with only local scope. Thus, it is the job of the REST run-time system to transform references to local state variables into references to the global state vector. This is accomplished by requiring the user to call REST run-time routines `GetState(Arg)` and `PutState(Arg,Value)` in order to read or write local variable `Arg`. These routines use the run-time system's knowledge of which module variable is making the

call, and translate the reference into the correct global state vector element representing the module's own `Arg` value. The REST run-time system is also responsible for all analysis, and sequencing of routines declared in the RML modules.

A legal module type definition is given in Figure 2.1. The example is somewhat artificial, in order to illustrate in one place several different RML features. Development of a more complete example is found in Chapter 3. Later sections will outline the specific meanings of various parts of the definition, nevertheless, the basic description can be understood without the details. The example describes a hardware component that interfaces with two hardware "channels". The interface may fail, then repair, as may the channels. The interface notifies another portion of the system whenever it fails or repairs; a channel notifies the interface whenever the channel fails or repairs. The interface keeps track of the operational status of its two channels. The entire system is considered to have failed if it is ever the case that any module of type `InterfaceChannel` and both of its channels are simultaneously in failed states.

Each of the RML portions will now be described in turn, beginning with module type definitions. Following that, we describe the RML translator, and REST run-time system.

2.2 MODULE Definitions

A module type definition always begins with the keyword MODULE, followed by a module type "identifier" (as in most programming languages, an alphanumeric character string beginning with a letter) and a '{.' A module type identifier corresponds to a user-defined type name in traditional programming languages. A module type definition has three parts: variable declarations, event declarations, and action code. The module type definition is terminated with a } that matches the leading { . The variable declaration section holds declarations of STATE variables, RATE variables, and RELATION variables. Its various components will be discussed at length, beginning with STATE variable declarations.

2.2.1 STATE Variable Declarations

A STATE variable declaration has the form

```
STATE StateVariableName;
```

where `StateVariableName` is an identifier. A STATE variable is implicitly an integer, and has the same storage size as an ordinary C integer variable would have on the target execution engine. One-dimensional arrays may also be defined as STATE variables. Several STATE declarations may be placed on one line. For example, the following is a legal STATE declaration

```
MODULE InterfaceChannel {
    STATE OpState;          // bit j set -> Ch[j] is dead, j=0,1
    RATE  ICFailureRate,ICRepairRate;
    RELATION Channel CH[2] : RecvCHMsg();
    RELATION GENERIC Sys ;

    // Constants          FAILED = 4, ALIVE = 8
    // are declared in the STATIC section of REST model
    //   In C  ! is logical negation, ^ is bit-wise exclusive OR
    //         | is bit-wise OR      , & is bit-wise AND
    //         || is logical OR      , && is logical AND

    //  The interface fails
    FailEvt:IF(!(GetState(OpState) & FAILED)) TRANTO
              { int OS = GetState(OpState);
                   PutState(OpState,OS|Failed);
                   SendValue(Sys,FAILED);
               } BY ICFailureRate;

    // The interface repairs itself
    RepairEvt: IF( GetState(OpState) & FAILED) TRANTO
                { int OS = GetState(OpState);
                     PutState(OpState, OS &~FAILED);
                     SendValue(Sys,ALIVE);
                 } BY FAST ICRepairRate;

    // The system is in a failure state
    DeathEvt: DEATHIF ( (GetState(OpState) & FAILED) &&
                        GetState(CH[0].Status) == FAILED &&
                        GetState(CH[1].Status) == FAILED );

    CODE { void RecvCHMsg(msg,who)
                int *msg,who;
               {int OS = GetState(OpState);
                // create bitmask describing caller
                int WhoBit = 1 << who;
                    if(*msg==FAILED)   // record failure
                      PutState(OpState, OS|WhoBit);
                     else              // clear failed bit
                      PutState(OpStat, OS & ~WhoBit); }
               } // end of CODE
}               // end of MODULE
```

Figure 2.1: Example of REST Module declaration

```
STATE Status, Mode, InputPorts[4];
```

Here STATE variables **Status** and **Mode** are declared, as is an array **InputPorts** having four elements. As in C, array indexing begins with 0. Multiple lines of STATE declarations are permitted.

2.2.2 RATE Variable Declarations

A RATE variable declaration gives a symbolic name to a transition rate, and is given with the intention of that rate being referenced later in the action declaration part. A RATE variable is implicitly a floating point number; unlike STATE variables, arrays of RATE variables are **not** permitted. The following is a legal declaration of RATE variables **FailRate** and **RecoveryRate**

```
RATE FailRate, RecoveryRate;
```

Values for RATE variables are **not** declared within the module definition. Like STATE variable declarations, a module may contain a number of lines of RATE variable declarations.

It is also possible to declare a RATE **function**, e.g.,

```
RATE FailRate();
```

In this case **FailRate()** is a floating-point valued function whose body is given in the module's action description part. The effect is to use the value returned by the function in place of a static value bound to an ordinary RATE variable. A RATE function declaration must appear on its own line.

2.2.3 RELATION Variable Declarations

A RELATION variable is a reference (or pointer) to another module with which a module of the type being defined may be "connected" via the **Connect()** call (see Section§2.5). In Figure 2.1 the declaration

```
RELATION Channel CH[2] : RecvCHMsg();
```

means that each module of type **InterfaceChannel** may be logically related to two modules of type **Channel**. Within the scope of the **InterfaceChannel** definition, array name **CH** refers to these two modules. Examples of this are seen in the arguments of the **SendValue()** function calls. It is important to remember that these declarations refer to structural properties that all instances of modules having type **InterfaceChannel** share. Instantiation, initialization, and interconnection of module variables are made elsewhere in a REST model.

A module is able to communicate via message-passing with its RELATIONS. The `SendValue()` calls are examples of an `InterfaceChannel` module sending messages (in this case, single integers) to a RELATION. Function `RecvCHMsg()` within the RELATION declaration specifies that whenever the modules bound to `CH[0]` or `CH[1]` send a message to a `InterfaceChannel` module, recognition and receipt of the message is accomplished by execution of routine `RecvCHMsg()`, expressed within the `InterfaceChannel` definition. An extended discussion on message-passing semantics is found in §2.7. Declaration of a message-handling routine may be omitted when messages are not expected from the RELATION. A run-time error will occur in the event that a message is sent to module through this RELATION.

When declaring a RELATION variable the variable's module type (e.g., `Channel`) may be omitted. In this case the keyword GENERIC is used in place of the module type; see, for instance, the declaration of RELATION variable `Sys` in Figure 2.1. RML permits a module to read the STATE variables of a RELATION, provided the RELATION's type is given. An example of this in Figure 2.1 is the function call `GetState(CH[0].Status)`. `GetState()` is a REST run-time routine used to access STATE variables; the syntax `CH[0].Status` means "STATE variable `Status` from the module connected as `CH[0]`". Since STATE variables are declared only in the context of MODULE declarations, it is feasible to access `CH[0].Status` only if we know that modules connected to `CH[0]` must have a STATE variable named `Status`. However, there is also an advantage to declaring GENERIC relations when a module definition is to be included in a library, or when modules of a given type are connected to different types of modules in one system. Since the type of module bound to a GENERIC RELATION is unknown and not necessarily constant, REST does not permit attempts to read STATE variables of such relations.

2.2.4 Event Declarations

REST's notion of an event is derived from the ASSIST declarations of DEATHIF, TRANTO, PRUNEIF, and ASSERT declarations [1]. Each of these ASSIST declarations can be viewed as an assertion "If *Condition* is true, then undertake *Action*". In the case of all these declarations, *Condition* is some logical condition of the system state vector; for DEATHIF the *Action* is "record current state vector as a death-state", for PRUNEIF the *Action* is "prune further modification of the state vector", for TRANTO the *Action* is modification of the state vector in accordance with rules specified following the TRANTO statement. In RML we add the EVENTIF event type. EVENTIF is an event where a logical condition of the system state vector is specified, but the only action taken is to record that the event occurred. RML's EVENTIF is just the converse of ASSIST's ASSERT statement, which specifies a logical condition and notifies the user whenever the condition is **not** satisfied.

In RML we view each of these as specific event types. RML then permits the user

to label event declarations, e.g. labels **FailEvt**, **RepairEvt**, and **DeathEvt** in Figure 2.1. Event labels play an important role in identifying event declarations in RML translator error messages, run-time event logs, and in the REST debugger.

The DEATHIF declaration in Figure 2.1 asserts that a system failure condition occurs whenever the **Status** variables of both **CH** RELATIONS connected to an **InterfaceChannel** module are equal to the constant **FAILED** at the same time as the module's **OpStatus** also reflect failure. It is important to remember that this DEATHIF statement applies to all modules of type **InterfaceChannel**. An operational system fails whenever this condition becomes true for any **InterfaceChannel** module.

RML permits the syntax

```
DEATHIF DeathCondFunction();
```

where **DeathCondFunction()** is an integer valued C function which is declared later in the module's CODE section. This function is called to evaluate whether the system state is in a specific failure condition, in which case a non-zero value is returned. Value zero is returned to indicate that the failure condition does not exist in the system state vector. Any function so used must be expressed in the CODE segment of the module being defined. RML permits the function to be declared with arguments, e.g.,

```
DEATHIF DeathCondFunction(1,GetState(OpState));
```

is legal. Of course, the user is responsible for making sure that the arguments match in type and number with those in the function declaration.

All RML event declarations may express *Condition* either as a logical C expression within matching parenthesis, or as a call to an integer function. However, there is a very important difference between the two statements

```
DEATHIF DeathCondFunction();
```

and

```
DEATHIF ( DeathCondFunction() );
```

The first implicitly asserts that within the CODE segment of the MODULE declaration there will be an integer function called **DeathCondFunction()**. Presence of the parenthesis in the second form is a signal to the translator to treat the text within the parenthesis as a C expression. Any function that appears in this expression is *not* declared in the module's CODE section. Instead, it may be declared in the STATIC section of the REST model (see §2.3). The utility of the second form lies in the fact that functions in STATIC may be referenced from within any MODULE definition in the model. The reason for the

difference in these two forms lies in the way the RML translator works. When the first form is recognized, the translator modifies the name of the function by prepending the module type name, in order to make the function name unique. This is necessary because the scope of a function declared inside of a CODE section is limited to that section. Therefore, two different MODULE definitions can use the same name for two semantically different functions. Any function expressed within the CODE segment of a module also has its name modified. Ultimately, the first form of the DEATHIF is transformed into a C statement such as

```
if( InterfaceChannel_DeathCondFunction() ) {   .
```

When processing the second form, the RML translator does not prepend the module type to function calls it encounters; the second form is translated into a C statement such as

```
if( DeathCondFunction() ) {   .
```

It is also permissible to define a macro in the STATIC section, and use it with the second form. For example, if STATIC contains the macro definitions

```
#define EQS(a,b) (a==b)
#define SET(a,b)  (a&b)
```

then the DEATHIF statement in Figure 2.1 could be replaced by

```
DEATHIF( SET(GetState(OpState),FAILED) &&
         EQS(GetState(CH[0].Status),FAILED) &&
         EQS(GetState(CH[2].Status),FAILED) );
```

As the STATIC section is intended for C statements common to all modules, calls to REST run-time routines (such as `GetState`) may not be placed there, as the arguments to such have scope limited to a module's definition.

The PRUNEIF and EVENTIF event syntax is identical to the DEATHIF syntax. A recognized PRUNEIF condition terminates further transformation of the current system state. A recognized EVENTIF declaration does nothing except call a log routine which records the occurrence of the event.

System state transitions are also events, of the form

IF *Condition* TRANTO *Action* BY *Rate*;

As illustrated by Figure 2.1, *Action* may be any C *compound statement*[2]. A compound statement begins with a '{' after which it may declare variables with scope local only to the compound statement, followed by a sequence of statements and terminated by a '}'. Observe

that variable `OS` is defined within the *Action* portions of both the FailEvt and RepairEvt events in Figure 2.1. Function calls within a complex C statement must either be to REST run-time routines, or to functions defined in the model's STATIC section. They specifically may not be to functions declared within the CODE section. Alternatively, *Action* may be a call to a C function (of type void), e.g.,

```
IF( GetState(Status) == ALIVE) TRANTO FailMe() BY FailRate;
```

Similar to condition functions, `FailMe()` may include arguments. Also like condition functions, this form of the statement implicitly asserts that there is a function named `FailMe()` declared in the module's CODE section.

The rate variable following the keyword `BY` must be declared in the module as a RATE variable. A declared rate function may also be used, e.g., we may replace `FailRate` with a function call `FailRate()`. Like the other function calls, this one is permitted to have arguments (although the arguments must not be declared in the RATE variable declaration itself). This is particularly useful for implementing state-dependent transition rates. As in ASSIST, the keyword FAST signifies an exponentially distributed recovery transition.

2.2.5 CODE Segment

As we have seen, C functions can be specified to handle message receipt from RELATIONS, to compute whether an event's *Condition* is satisfied, to implement effects of a satisfied TRANTO *Condition*, and to return a transition rate. All functions so declared within a MODULE definition must be expressed in the CODE segment of the MODULE definition. These routines may call REST run-time functions such as `GetState`, `PutState`, `SendMsg`, `SendValue`.

Routines in the CODE section must follow the following rules.

1. *Condition* calculation routines must return an integer value. Routines `PutState()`, `GetNewState()`, `SendMsg()`, and `SendValue()` may not be called.

2. *Action* routines must be of type void.

3. Rate calculation routines must return a floating point number. They may not call `PutState()`, `SendMsg()`, or `SendValue()`.

4. Message handling routines must be of type void. Furthermore, the first argument (if any) must be a character pointer. A second argument, if any, must be an integer.

The first argument passed to a message-handling routine is a pointer to the message. The second argument is the array index of the calling RELATION. Both of these facts are

illustrated in Figure 2.1, where **msg** points to the integer-valued message, and variable **who** is 0 or 1, depending on whether `CH[0]` or `CH[1]` sends the message.

It is possible to put commonly used subroutines in the CODE segment, but care must be taken to remember how the RML translator works. Every function found declared in a CODE segment has its name modified by pre-pending the module type name and an underscore. Functions that are *called* from functions in a CODE segment do not have their names modified. So if a subroutine is declared in the CODE segment, e.g., routine `Foobar()`, and that subroutine is then referenced by using its modified name, e.g., `IC_FooBar()`.

At present there are no safety mechanisms in REST to prevent a function in one module's CODE segment from using this subroutine trick to call a function declared in another module's CODE segment. However, this is not recommended programming practice!

2.3 STATIC Section

A STATIC section optionally follows the set of all MODULE declarations. It is declared with the keyword STATIC followed by a bracketed section. This section is designed to hold any and all C language functions, macros, struct definitions, and data structures used to support the analysis of the REST module. Anything declared in the STATIC section can be accessed by any module code. We have already seen the example of STATIC containing macros which are used in event conditional statements. A good use of STATIC is to create and store complex data-structures that do not change as the model is analyzed. For example, a data structure may be defined which describes the static topology of a network. Once it is initialized (within the INIT section, to be described), any REST routine may read the data structures. For that matter, any REST routine can *write* the data structures, but this is not recommended unless the user is highly knowledgeable about his model and the way REST analyzes it. Due to the assumed invariant nature of the data structures declared in the STATIC section, a variable declared in STATIC may not play the role properly reserved for STATE variables.

STATIC may not contain any reference to local variables in modules, or to REST run-time routines **GetState**, **PutState**, **GetNewState**, **SendMsg**, or **SendValue**.

2.4 GLOBAL Section

Instances of module variables are declared in the GLOBAL section. This section is introduced by the keyword GLOBAL, and like STATIC is delimited by left and right braces. GLOBAL consists of a sequence of declarations of the form

```
ModuleType VariableName;
```

or

```
ModuleType VariableName[Integer];
```

This declares there is a module variable named **VariableName**, of type **ModuleType**, which must be defined earlier in the REST module. An array of variables is declared using the second form; **Integer** must be a positive integer—symbolic names via #define are not permitted.

2.5 INIT Section

The INIT section contains code to create the communication topology between declared module variables, and to initialize the state variables of the modules. The INIT section is, without question, the hardest part of a REST module to follow, and debug. One of the great benefits of a graphical front-end to REST will be the automated generation of INIT section statements. Like other REST sections, the INIT section is declared using a keyword INIT, and is delimited by left and right braces. The INIT section has all the properties of the body of a C subroutine, which it does, in fact, become in the C translation of RML. The braces delimiting INIT's scope are the braces delimiting the subroutine. Thus temporary values, macros, and so forth can be defined at the beginning of the INIT section, just as done in a C routine. The remainder of the section uses a C framework to call REST routines **Connect()**, **View()**, **Set()**, and **Rate** to initialize the model. Each of these is now described in turn.

A **Connect()** call establishes a correspondence between two module variables. It makes real the connection implied by a RELATION variable declaration. The call has the form

```
Connect(ModuleVar1.RelationVar1, ModuleVar2.RelationVar2);
```

Here, **ModuleVar1** specifies some specific module variable declared in the GLOBAL section. If the variable is a member of a declared array, then the array index must be given as well, e.g, **FTPChannel[3]**. Character string **RelationVar1** must be the name of a RELATION variable declared in the definition of **ModuleVar1**'s module type. If that relation was declared as an array, then **RelationVar1** must include a specific array index. These same rules apply to **ModuleVar2** and **RelationVar2**.

The **Connect()** call asserts that **ModuleVar2** is the instantiation of the module type referenced as RELATION **RelationVar1** in module **Module1**, and that conversely **ModuleVar1** is the instantiation of the module type referenced as RELATION **RelationVar2** in module **Module2**. It gives substance to the RELATION abstraction.

It can be useful to use C expressions, functions, and loop constructs to express variable or relation indices. For example, the loop

```
for(idx=0; idx<10; idx++) {
    Connect(FTP[ (int)(pow(2,idx))%17].Ch, FTPC.Boss[16*idx/137]);
```

is perfectly legitimate, provided that all indices so computed are within range of the declared arrays (this is checked at run-time), and "duplicate connections" are not made (which is also checked).

It is sometimes useful to endow one module with the ability to "view" another's STATE variables with no intention of message communication to or from that module. To support this situation REST provides a routine `View()`, whose arguments are identical to those for `Connect()`, save that no `RelationVar2` is specified. The effect of call

```
View(ModuleVar1.RelationVar1, ModuleVar2);
```

is to enable `ModuleVar1` to access `ModuleVar2`'s state variables via RELATION variable `RelationVar1`. The connection goes in only one direction. The definition of `ModuleVar2`'s module type does not need to declare a RELATION variable with `ModuleVar1`'s type. For all practical purposes, `ModuleVar2` is ignorant of the fact that `ModuleVar1` has access to it.

The `Set()` call is used to initialize state variables. It has the form

```
Set(ModuleVar.StateVar, expression);
```

where `ModuleVar1` specifies a module variable, `StateVar` specifies a single state value (possibly indexed), and `expression` is a C expression that evaluates to an integer. For example, the following C fragment is a convoluted way to initialize the STATE variable `OpState` of module `IC[3]` having type `InterfaceChannel` (recall Figure 2.1) with value 0.

```
Zero = 0;
Ones = -1;
Set(IC[3].OpState, (Zero|Ones) & Zero);
```

The `Rate()` call is used to initialize `RATE` variables declared within MODULE declarations. It has the form

```
Rate(ModuleType.RateVariable, float);
```

where `ModuleType` is the name of a declared MODULE type (not variable), and `RateVariable` is the name of a `RATE` variable declared by that MODULE. `float` is a floating point number, used as the transition rate by any TRANTO statement using variable `RateVariable`. It is important to remember that all module variables of type `ModuleType` use a common value of `RateVariable`. We intend to permit variable-by-variable declarations of rate variables in later versions of REST.

As with the other REST initialization routines, any module or relation index can be given as a C expression which evaluates to an integer.

2.6 STATE Variable Access

REST provides routines `GetState()`, `GetNewState()`, and `PutState()` for reading, and writing the system state vector, respectively. `GetState()` and `GetNewState()` both have a single argument that uniquely identifies one state value. That argument may describe one of a module's own state values, e.g., `GetState(OpState)` in Figure 2.1. Alternatively it may describe a state variable for one of its relations, e.g., `GetState(CH[0].Status)`, also in Figure 2.1. Of course, in the latter case the relation must be identified. In either case, any index may be expressed as a C expression, which will be evaluated at run-time.

A call `PutState(StateVar,Value)` identifies one of a module's own state variables, and the value to give it. `Value` may be any C expression that does not include a call to other REST runtime routines. Observe that while a module can *read* its relation's state variables, it is unable to *write* them.

The difference between `GetState()` and `GetNewState()` lies in *which* state value is returned. `GetState()` always returns a value from the "base" state vector, the one whose elements are tested in IF events for transition conditions. `GetNewState()` can be useful in the processing associated with the *Action* portion of an IF event. Some portion of that processing may modify a state variable via a `PutState()` call. Later processing—within that same *Action* processing—may want to access the modified variable instead of the "original" variable. `GetNewState()` provides that function.

2.7 REST Message-Passing

As we have seen, an RML model is very modular; expression of transition rules and recognition of failure conditions are all accomplished by referencing local variables. In addition, RML follows an object-oriented philosophy by using explicit message-passing to express information transfer between modules. The correctness of an RML model depends on the model developer's understanding of the semantics of message-passing. This section directly addresses message-passing syntax and semantics.

Module variables may communicate with each other during *Action* processing via message passing. A typical example is the *Action* processing associated with a component failure. The module variable representing the component executes its *Action* code, which may send a message to all modules in which the component appears as a relation. The message handling routines in recipient modules are executed; in that execution, additional messages may be sent and received. To use message passing properly, it is important to understand exactly how message passing is implemented. Suppose `SendMsg(RelVar,Msg)` is called from within some *Action* code, with the recipient relation specified. `SendMsg()` looks up the module

and relation to which **RelVar** has been Connected. Part of **Connect** processing is to store a pointer to the function declared to handle messages from the sender. This pointer is accessed, and the function is called. Thus, a call to **SendMsg()** essentially is a call to a subroutine to handle the message sent. Control is not returned to the calling code until all processing associated with handling that message has completed. Remember, however, that a message handler can itself call **SendMsg()**; this arrangement uses the C run-time stack to manage multiple message invocations—we simply transform message passing into subroutine calls.

REST actually provides two routines for sending messages. **SendMsg()** is the more general of the two. A **SendMsg()** call has two arguments. The first identifies the RELATION variable to whom the message will be sent, the second is a pointer (of type (char *)) to a message. The programmer is responsible for managing the space used to store the message, and for building the message itself. The space may be acquired from **malloc()**, or may be part of the calling routine's local space. The space should **not** come from data structures declared in the GLOBAL section. It is good programming practice to have the routine that calls **malloc()** also call **free()**. **SendValue()** is similar, except that the second argument is an integer *value*. When applicable, this routine relieves the programmer from memory management. **SendValue()** itself allocates space, copies in the value, and passes a pointer to that value to the appropriate message receipt function.

Chapter 3

Extended Examples

This chapter presents a series of examples given in both RML and ASSIST. These examples illustrate both the use of RML, as well as its relative advantages and disadvantages with respect to ASSIST.

3.1 Example 1: A Non-Reconfiguring Quad

The non-reconfiguring quad is a simple system consisting of 4 identical processing components which cooperate through majority voting to achieve greater reliability. Since the system does not disable channels on its voters (reconfigure), any two processor failures have the potential to defeat the majority vote. The ASSIST model for this system is shown in Listing 1 and the RML model in Listings 2a-b, found in Chapter 5.

The ASSIST Model: In the ASSIST model, an *ARRAY* of 4 processing elements, P, is declared in the *SPACE* statement. The processing elements have binary state values where 0 represents GOOD (healthy or working) and 1 represents BAD (failed or not working). The 4 processors are initialized to GOOD in the *START* statement. System failure is declared in the *DEATHIF* statement to occur when more than 1 processor has failed (more than one element of the processor array, P, is set to 1 so that the sum over the array is greater than 1). The system state matrix is described by a *FOR* loop which declares that, if a processor is GOOD, it can fail to BAD by rate PFail. The constant PFAIL is given a value of 1.0E-4 in the *ASSIST* preamble.

The RML Model: RML was designed to support a graphical model description environment where a user defines the reliability model as he constructs a system block diagram. To construct the block diagram, components are defined and interconnected. Component definition is thought of as the instantiation of a particular type of component. In this example we would have four instantiations of one type of processor component. To support this

paradigm, RML allows the user to define MODULEs which describe component types. The components themselves are declared in a GLOBAL section of the model and interconnected in the INIT section (see Listing 2b).

A module contains 5 kinds of entries: STATE variables, RATE variables, RELATION declarations, EVENT declarations and CODE. A module can have several STATE and RATE variables. The **Processor** module (Listing 2a), however, contains one STATE variable, **Procstate**, and one RATE variable, **ProcFail**. Following these is one TRANTO event. The TRANTO event in RML is identical in meaning to the ASSIST TRANTO statement with the exception that functions can be called in the condition, effect and rate fields of the TRANTO. The **Processor** TRANTO event uses the simple functions **ProcGood()** and **FailEffect()** to illustrate this capability. The functions are declared in the CODE section of the module and are written using standard C language syntax. Notice in the function definitions that the STATE variable **ProcState** is accessed through functions **PutState()** and **GetState()**. The modeler uses the TRANTO events (rules) and C code functions to define the behavior of the component. In the **Processor** module, this is simply, as before, that if a **Processor** is **GOOD** it will FAIL with a rate **ProcFail**.

RML models typically contain a **System** module which contains events describing system failure (DEATHIF events). A DEATHIF event identifies the state or set of states in which a subsystem is considered to be no longer functioning correctly. The DEATHIF expression is then written in terms of the subsystem's STATE variables. To do this, the **System** module must have access to the subsystem's internal STATE.

Following this line of thought, the **System** module in Listing 2a begins with a RELATION statement declaring that the **System** module can communicate with 4 **Processors**. The DEATHIF event references function **DeathCond()**. Function **DeathCond()** retrieves each **Processor**'s STATE, counts the number of failed **Processors** and returns **TRUE** if more than one **Processor** has failed.

The STATIC section of an RML model is used to declare constants and macros. In Listing 2b, two states are defined (**GOOD** and **FAIL**) and a constant, **NumProcs**, which refers to the number of **Processors** in the subsystem. (Note that, as of this writing, a constant such as **NumProcs** cannot be used in component array declarations. See next paragraph.) The macros **ISGOOD** and **FAILED** make the simple tests for the **GOOD** and **FAIL** state.

The GLOBAL section in Listing 2b creates 5 components, a set of 4 **Processors**, **P[4]**, and a **System**, **SYS**. The constant, **NumProcs**, could not be used to define the size of **P[]**, because these structures are not visible to the C code (thus the use of **GetState()** and **PutState()**).

The INIT section defines the interconnections between the declared components and initializes the components STATE and RATE variables. In Listing 2b, the 4 **SYS.Proc[]** variables are connected to the actual **P[]** components. The **View()** function defines a one-

way connection. A `Connect()` function would be used to establish a two-way connection. The `Set()` function initializes the 4 `P[].ProcState` variables to `GOOD`. The `RATE()` function initializes the single RATE variable associated with the `Processor` module.

3.2 Example 2: A Reconfiguring Quad

A reconfiguring quad provides greater reliability than a non-reconfiguring quad by disbaling a failed processing channel. The disabled processor is no longer included in the voting. The voter is thus protected against another failure. Reliability depends on how long it takes the system to detect the failed channel and disable it. The reliability model must now keep track of `GOOD` processors, `FAILED` processors and `DISABLED` processors.

The ASSIST Model: The ASSIST model is shown in Listing 3. A processor state is modeled as being either GOOD, BAD or neither. A processor that is neither GOOD nor BAD has been disabled. The SPACE statement defines two arrays to track the GOOD and BAD processors. The START statement initializes the processors to GOOD. The DEATHIF statement defines failure of the majority voter when the number of BAD processors is equal to or greater than the GOOD processors. The two TRANTO rules state that when a processor is GOOD it can fail to BAD with rate `ProcFail` and that when a processor is BAD it can be disabled by a FAST transition with rate `Prec`.

The RML Model: To model the reconfiguring quad in RML, we could place the majority vote test (which is in the DEATHIF statement in the ASSIST model) in the `System` model DEATHIF event. However, we wish, with this model and the next, to begin to take steps towards model structures which more closely simulate system function. To this end, we introduce an `FTP` module which represents the redundancy management process of the fault-tolerant computer. The FTP module is thus a logical component. It is the FTP module's responsibility to detect failed processors and disable them. The FTP module also tests for health of the quad with the majority vote rule.

The RML model is shown in Listings 4a-e. The `Processor` module has been augmented to include a RELATION with the `FTP`. The `Processor` can be disabled by the `FTP` through the `FTPmess()` function. A third state value, `REMOVED`, has been added to accommodate the disabled processor.

The `FTP` module, Listing 4b, has a reciprocal RELATION with the `Processor`. The `FTP` module TRANTO statement states that if the `FTP` is `VULNERABLE` it will `RECOVER` by a FAST transition of rate `FTPrec`. The vulnerable state is defined in the function `Vulnerable()` as when one or more of the `Processors` is failed. Recovery is implemented by sending failed `Processors` the message that they are now `REMOVED` (see Listing 4c for `Recover()` function).

After a recovery or a receipt of a `Processor` message, the `FTP` must re-evaluate its health.

This is done in the function **Eval()** (Listing 4b). In **Eval()** the number of **GOOD** and **BAD Processors** are counted. The **FTP** state is set to **GOOD** if there are no **BAD Processors** and **FAIL**ed if the number of **BAD Processors** is equal to or greater than the number of **GOOD Processors**. Notice that this is precisely the same test as was done in the ASSIST DEATHIF statement.

The **System** module simplifies to a death condition which returns true when the **FTP** (Listing 4d) has failed. An **FTP** is declared in the GLOBAL section and **Connected** to the **Processors** in the INIT section. The **System** module **Views** the **FTP** (Listing 4e).

3.3 Example 3: A Reconfiguring Quad with Voter

The previous examples lack physical interconnections and, as such, do not give the reader an appreciation for the added detail of a graphically derived RML model. A quad fault-tolerant computer would have interconnections between the voters of the processors. A block diagram might be drawn as shown in Figure 3.1. Here, a processor is composed of a **CPU** and a **VOTER**. The **CPU** communicates with the **VOTER** over a bidirectional link. Data sent from the **CPU** to the **VOTER** is relayed by the **CPU**'s local **VOTER** to the remote **VOTER**s. Assuming a degree of synchronization between the processors, all **VOTER**s will have 4 copies of the data that they can now vote, returning the result to the **CPU**.

An RML model which corresponds to the block diagram of Figure 3.1 is shown in Listing 5. In the **Processor** module (Listing 5a) a RELATION to the **Processor**'s local **Voter** (VME) has been added. The **Voter**'s message receiver is **Vmess()**. In this model, when a processor fails, it has the additional effect of propagating the errors to the local **Voter** via **SendValue(VME,ERROR)**. Conversely, a failed voter causes the **Processor** to fail through the **Vmess()** function. Notice that the **FTP** module no longer sets the **Processor** to the **REMOVED** state. In this model a **Processor** is disabled in the **Voter** module as will be discussed next. Also note that the test for **GOOD** has been written without a function call in the TRANTO and that the test has changed from an equate to a bit test. As the model becomes more complex, multiple state descriptors will be used (see the section on I/O interface). Having each state descriptor assigned a bit value (i.e., 1, 2, 4...) facilitates defining and testing states.

The **Voter** is modeled with 3 state variables: the **Voter** state (e.g., **GOOD, ERROR** or **FAIL**), a bitmap of **Voter** errors (bit 0 set if **Processor** 0 sends errors) and a second bitmap (**VoterEnable**) enables **Processor** channels into the **Voter**. (See Listings 5b-c). The last two state variables correspond to the error register and enable register which might be found in an actual vote circuit. A **Voter** has a RELATION with its local **Processor** and with the remote **Voters** as indicated by the block diagram. The **Voter** can send and receive

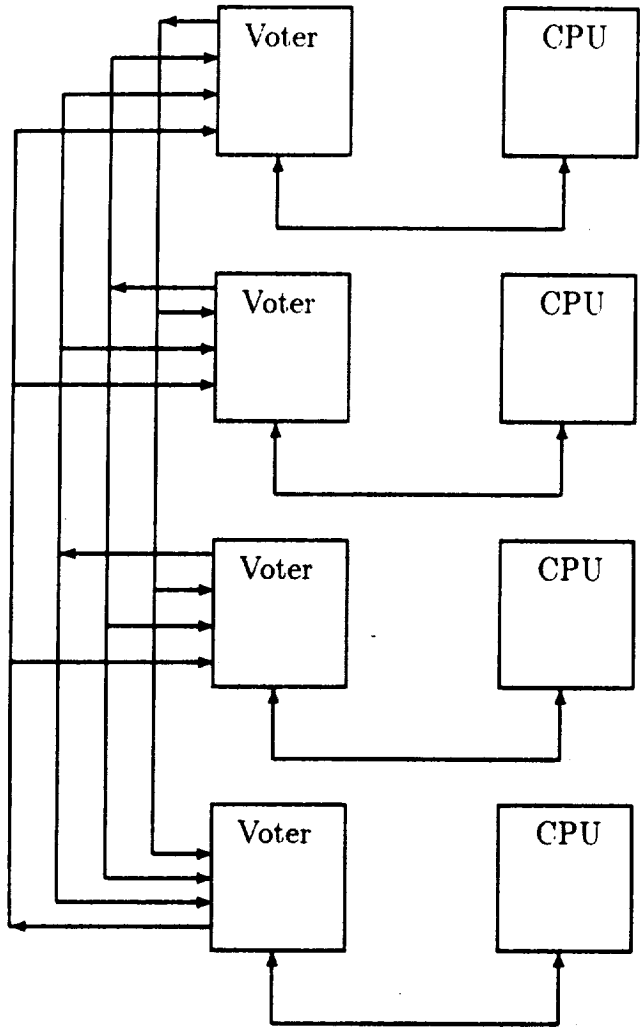

Figure 3.1: Block Diagram of Quad with Voter

messages to and from the other **Voters** in the system (including itself). In Figure 3.1 a **Voter**'s connection to the other **Voters** is drawn as a multi-drop, or broadcast bus. This type of interconnect has not yet been implemented in RML and therefore must be modeled as a fully connected interconnect, i.e., 4 transmit and 4 receive ports. The **Voter** must also communicate with the **FTP** so that the **FTP** can "read" the **VoteError** data (to detect errors) and set the **VoteEnable** state (to reconfigure).

As modeled here, the **Voter** is a logical component and therefore does not have a failure related TRANTO statement. However, a **Voter** can still fail when it is overwhelmed by bad data, i.e., the number of erroneous data channels are greater than or equal to the number of valid data channels. The **Voter** Module's **FailEffect()** function simulates **Voter** failure by changing the voter state to **FAIL** and sending a **FAIL** message to the local **Processor**. Because the **Voter** is a logical component, it doesn't originate **ERROR** messages. It does, however, route **ERROR** messages received from the local **Processor** to the other remote **Voters**. This is done in the **Pmess()** receiver function. The **Voters** are inhibited from sending the **ERROR** message in **Pmess()** when the **Voter** is FAILed. This is done to protect against an infinite loop between the **Voter** and its **Processor**.

The re-reouted **ERROR** messages are received by the **Vmess()** function of the remote **Voters**. In the **Vmess()** function, a bit is marked in the **VoterError** state variable which corresponds to the **Voter** which sent the **ERROR** message. This bit is only marked if the sender was enabled. Since there might have been a state change, the **Eval()** function is called.

The **Eval()** function uses the **VoterError** and **VoterEnable** state variables to determine the **VoterState** (**GOOD**, **ERROR** or **FAIL**) (remember that the bits in the **VoterError** and **VoterEnable** state variables correspond to the **Voter**'s input channels.) The evaluation function is, as before, the majority vote rule. The **Eval()** function, looking only at enabled input channels, counts the number of channels with and without errors. If there are any enabled channels with errors the **VoterState** is set to **ERROR**. If the number of channels with errors is equal to or greater than the number of channels without errors the **Voter** fails.

During the recovery process, the **FTP** module will "read" the **VoterError** state and construct a new **VoterEnable** state to disable failed **Processors**. The **FTP** sends the new **VoterEnable** state to the **Voter** module through the **FTPmess()** receiver function. The **FTPmess()** function loads the new **VoterEnable** state and then clears the **VoterError** state from any errors that may have been caused by **Processors** which are now disabled. Note that if a **Processor** was producing errors and the **FTP** did not correctly configure the **VoterEnable** state to mask this **Processor**, then the errors would still be present. Finally the **VoterState** is re-evaluated.

With the addition of the **Voters**, the simulation of the recovery process has become more complex. The **FTP** module (Listing 5d-e) must now maintain a **WorkingSet** state variable

which is identical to the `VoterEnable` state variable, i.e., it keeps track of which `Processors` are part of the active configuration (or enabled into the system's voters). While this may seem to be redundant, it actually models quite closely how a real system might operate.

The `FTP's Eval()` function has been simplified because the `Voters` now perform the majority check. An `FTP` is now defined as failed if there remains no `GOOD` Processors. The `Vulnerable()` function has been modified to return true if any `Voters` in the `WorkingSet` have detected an error.

The `Recover()` function has become very complex. Each `Voter` in the system maintains error data on all the `Voters`. To diagnose the system, it is not enough to sense whether or not the `Voter` has had an error, but it must be determined on which channel the error occurred. Additionally, a system wide consensus must be arrived at, i.e., more than one enabled `Voter` must agree that a particular `Processor` is faulty before that `Processor` is disabled. To accomplish this, the `Recover()` function keeps four error counters (`P1e..P4e`). Each enabled `Voter`'s `VoterError` state is read and passed to a subroutine (`Counts()`) where the bits are tested for errors. If a bit is set in the `VoterError` state variable, the corresponding error counter is incremented. An error count greater than 1 indicates that at least two enabled voters detected an error on the corresponding `Processor`. The appropriate bit is set in another variable, e, if a `Processor` failure is diagnosed. Those faulty `Processors` are then removed from the `WorkingSet`. The new `WorkingSet` is then sent to the `Voters` for storage in their enable registers. Finally, the state of the `FTP` is re-evaluated.

3.4 Example 4: An Redundant I/O Interface

This section defines a quad redundant I/O interface. The model is similar to the quad processor with the exception that only two interfaces are used at any time and only one is needed for safe operation. There are then initially 2 spare interfaces. This model is of interest because of the sequence dependency introduced, that is failure of a spare I/O interface has no effect on the system until one of the prime interfaces fail.

The ASSIST Model: When modeling the quad FTP two boolean state variables were kept for each processor to track a processor being GOOD, BAD or REMOVED. The quad interface requires that we keep track of a fourth attribute, i.e., whether or not the interface is currently controlling the I/O devices. In the ASSIST model shown in Listing 6, three boolean state variables are used. IGOOD[] indicates whether an interface is good or failed. A one in IO[] means that that interface is currently controlling I/O operations. If an interface has a detectable error (i.e., has failed while controlling I/O operations) than the corresponding IOError[] state variable is set to one.

The START state is initialized with 4 GOOD interfaces, interfaces 1 and 4 as controllers

(interfaces 2 and 3 spare) and no errors. System death occurs when there are no I/O controllers.

Two failure TRANTOs are written for each interface to capture the sequence dependency. When a GOOD interface fails while its controlling I/O, it causes an error. Six recovery TRANTOs describe the sequence dependent repair actions. For example, if interface 1 fails and interface 2 is still good, then interface 2 is enabled as a controller.

The RML Model: The RML model shown in Listings 7a-e, begins with an `Interface` module which is similar to the `Processor` module. Like the quad `Processors`, the `Interfaces` have a redundancy manager (`InterfaceManager`). The `InterfaceManager` can either `REMOVE` the `Interface` or place it `INUSE` (see function `Manmess()`). An `Interface` can thus be `GOOD`, `GOOD` and `INUSE`, `FAIL`ed or `FAIL`ed and `INUSE`. This conjunction of state descriptors is facilitated by using powers of two (1, 2, 4,...) for all descriptor values. Individual descriptors can then be ORed into or ANDed out of the state. For example, in the `Interface FailEffect()` function, the `Interface` state is changed from `GOOD` to (`FAIL`ed and `ERROR`s) by ANDing the state with the inverse of `GOOD` (thus removing the `GOOD` bit) and ORing in the `FAIL` and `ERROR` bits. Likewise in the `Manmess()` function an `Interface` is activated by ORing in the `INUSE` bit and deactivated by ANDing the state with the negative of `INUSE`.

The `InterfaceManager` module is similar to its `FTP` counterpart. Either the receipt of a message from an `Interface` (indicating that it has changed state) or the execution of the recovery TRANTO will cause the `InterfaceManager` to re-evaluate itself. The `InterfaceManager` is `VULNERABLE` when any `Interfaces` which are `INUSE` have errors. More to the point, an `InterfaceManager` cannot detect a failed `Interface` which is not `INUSE`. The `InterfaceManager` fails when there are no good `Interfaces` available. Recovery is accomplished by first sending the `REMOVED` message to those interfaces that are both `INUSE` and have `ERROR`s and then by setting to `INUSE` the spare `Interface`, if available.

3.5 Example 5: A Quad Processor with I/O Interfaces

In this section, the reconfiguring quad model is merged with the interface model. This is done to illustrate the utility of modular modeling. It is assumed that the failure dependency relation between a processor and its I/O interface is asymmetric, i.e., if a processor fails, the interface fails, but if the interface fails, the processor does not.

The ASSIST Model: The ASSIST model is shown in Listing 8. Note that the merge of the two models is in most cases accomplished by simplying combining fields in the SPACE, START, DEATHIF and TRANTO sections. An exception to this is the `Processor` failure TRANTOs. Due to the dependency of the interface on the processor, the failure effect of the

interfaces must be added to the processor TRANTO statements. Four processor TRANTOs are used to cover the the four interface conditions possible with the two boolean interface state descriptors Igood[] and IO[]. Note that in practice only three TRANTOs would be necessary since the state Igood[]=0 and IO[]=1 does not occur.

The RML Model: The RML model is shown in Listings 9a-k and a graphical depiction in Figure 3.2. Here the models are merged by first copying their respective modules into one file. Connecting the `Processor` to the `Interface` through a uni-directional link implies these modifications to the modules:

1. The `Processor` module is modified to contain a RELATION with an `Interface` and failure messages are sent to that `Interface` in the `FailEffect()` and `Vmess()` functions.

2. A reciprocal RELATION to receive the `Processor` message is defined in the `Interface` module. The `Pmess()` message receiver simply calls the `Interface FailEffect` function to propagate the effect of the `Processor` failure.

3. The GLOBAL, STATIC and INIT sections are merged.

4. `Connect()` statements must be added in the INIT section to define the `Processor-Interface` relations.

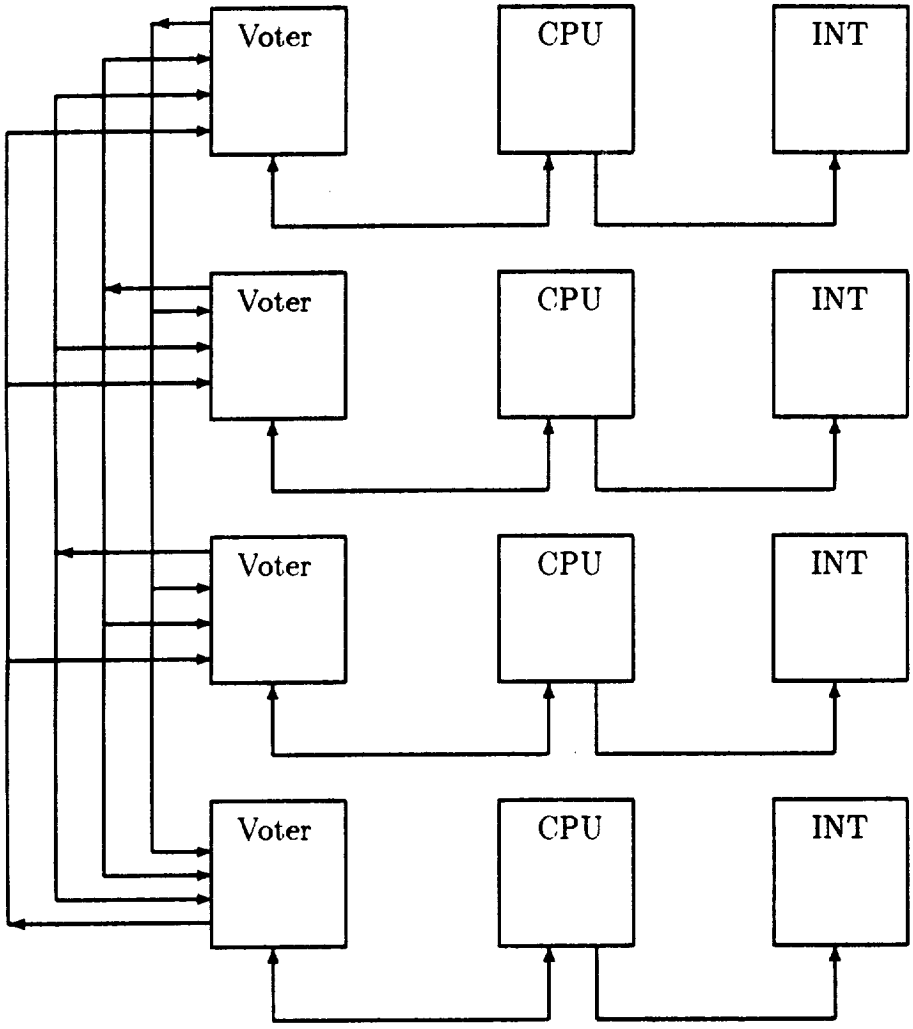

Figure 3.2: Block Diagram of Quad with Voter and I/O Interface

Chapter 4

XREST

This Chapter describes the X-Windows based user interface to REST, called **xrest**.

4.1 Building REST and xrest

The REST system comes in a tar file called rest.tar. To install REST, move rest.tar to a subdirectory that will be dedicated to REST, and type

```
tar -xf rest.tar
```

Following this, build REST by simply typing **make**. The make scripts take care of everything, but do generate a lot of messages along the way, especially when building the parsers for REST.

The main REST directory (where rest.tar was placed) is where the user interfaces with REST. In addition there are subdirectories obj and obj/working. Files in obj are related to the RML language translator and REST execution system. obj/working is a scratch directory that should have global read/write access.

4.2 Files required for compilation

The following files are either needed to build REST, or are provided in rest.tar.

- *Source code in the local directory:*
  Makefile, termlist, xfilt.c, xrest.c
- *Shell scripts in the REST directory:*
  BuildLocal, BuildRemote, KillLocal, KillRemote,
  ReRunLocal, ReRunRemote, RemoteSerial

- *Icon maps in the local directory:*
  addfunc.icon, addvar.icon, client.icon, connect.icon, filter.icon
  getst.icon, message.icon, rest.icon, run.icon, watch.icon

- *Sound files in the local directory:*
  comp_op.au, darth.au, flush.au, just_what_dave.au

- *Subdirectories of the current directory:*
  lib (REST module library directory),
  obj (REST local execution directory)

- *Example REST models in the* examples *directory:*
  example.rest, net.rest

- *In the /usr/include directory:*
  ctype.h, math.h, stdio.h, string.h, X11 subdirectory,
  xview subdirectory

- *In the /usr/include/X11 directory:*
  olgx subdirectory, X.h, Xlib.h

- *In the /usr/include/xview directory:*
  icon.h, notice.h, panel.h, rect.h, seln.h, sel_attrs.h,
  server.h, svrimage.h, termsw.h, textsw.h, xv_xrect.h, xview.h

- *Other files and programs needed:*
  /tmp, all the code associated with running a REST model

4.3 How to compile the program

Type **'make'** in this directory.

4.4 How to run the program

Type **'xrest'** while in X windows. All programs loaded and saved will be assumed to come from the directory from which you call xrest, unless explicit path names are supplied. If a file name is given on the command line, the file will be loaded in automatically. **xrest** will run under X windows releases 4 and 5.

4.5 About REST tool

When **xrest**, hereafter referred to as the REST tool program, is first started up, the main window appears. REST tool provides the following features, each of which is described in a subsequent section.

- Defined constraints
- The text editor window
- The REST message area
- Specifying names
- Stuff selected text
- Goto in the text window
- Set up a watch window
- Open a term window
- Run a REST model
- Module library
- Load a REST model
- Save a REST model
- Clear the template
- Quit REST tool

4.6 Defined constraints

The following constraints are *#define*'d at the beginning of the xrest.c source code program. Change them before compiling if needed.

- Maximum file name length = 128
- Maximum function name length = 24
- Maximum module name length = 24
- Maximum variable name length = 24

4.7 The text editor window

The text editor window, or the *template*, is located to the right of the main panel in the main window. It is the only window which enables the user to alter the current REST model. All intended modifications must be made in this window. All of the additional features [see sections §4.10, §4.11 & §4.12] work on the template.

In addition, the user can explicitly alter this area if so desired. Typing a key will cause its insertion at the caret; hitting the *Delete* key will remove the character to the left of the caret. The mouse buttons can also be employed in the window as follows:

- Click the **Left** button to move the cursor.
- Click and drag the **Middle** button to highlight text.
- Hold down the **Right** button for the text editor menu:
  - **File** - store is only option available
  - **View** - select line, line wrap
  - **Edit** - undo, copy, paste, cut
  - **Find** - find, replace
  - **Extras** - format features

There is also a vertical scrollbar which allows the user to move quickly through the REST model. Simply hold down the **Left** button in the direction of the desired move (up or down) and drag the mouse until the goal position has been reached. (Holding down the **Right** button on the scrollbar manifests some undocumented features which probably are of little use.)

A selection service is provided which allows the user to highlight text anywhere in a text or terminal window using the mouse. Click and drag the **Left** button to select text, then use the **Left** or **Middle** button to insert the selected text where desired.

To jump to the beginning or end of the REST model, or to a specific module, use the *Goto in the text window* button [see §4.14]. To jump to a specific *line* in the REST model, use the **View** option mentioned above.

The text window provides auto-indent when the user types directly in the window to modify the code. This feature causes a new line to be automatically indented to be lined up with the previous line whenever the *Return* key is pressed.

The following commands allow the user to manipulate the text window:

- *ctrl*-A = move caret to beginning of line

- *ctrl*-E = move caret to end of line
- *ctrl*-P = move caret up one line
- *ctrl*-N = move caret down one line
- *ctrl*-F = move caret one character forward
- *ctrl*-B = move caret one character backward
- *ctrl*-M = insert line
- *ctrl*-J = insert line (also)
- *ctrl*-I = insert tab
- *ctrl*-U = delete line
- *ctrl*-W = delete word (backward)

4.8 REST messages

The REST message area is located in the upper left hand corner of the main REST tool window. REST messages serve two purposes:

1. To inform the user of the last action performed.
2. To warn the user if the previous command could not be executed. This type of message is accompanied by a beep to alert the user of problems (names not specified, wrong names specified, etc).

A summary of the most frequently posted REST messages is given at the end of this documentation [see §4.24].

4.9 Specifying names

Three name request lines reside directly below the REST message area. Each seeks information from the user:

1. **Library dir:** the library directory name.

2. **File name:** the current REST model name.

3. **Module name:** the current module name.

To change any of these lines, **Left** click the mouse on the appropriate line at the spot where the caret should be placed and type away. The *Delete* key removes the character to the left of the caret, and *ctrl*-A and *ctrl*-E are used to move the caret to the front and back of the current line, respectively. Hitting *Return* moves the caret to the next line.

Whenever the user chooses a module library option [see §4.18], the library used is the one with the directory whose name is specified by the *Library dir* line. If no directory name is given, the current directory is used as a default.

Whenever a load or save option [see §4.19 & §4.20] is chosen, the name of the file read from or written to is the one specified by the *File name* line.

Whenever a module is added or added to [see §4.10, §4.11, §4.14, §4.15, & §4.18], the module affected is the one specified by module name. The exception occurs when a window names a module different from the one specified by the *Module name* line, in which case the module specified in the window is used instead.

4.10 Add a new module

This feature automatically inserts a new module into the REST model. To do this, simply click the left button to the *Module name* line [see §4.9], type in the desired module name, and **Left** click the *Add a new module* button. The new module will be inserted at the top of the REST model. If the module cannot be added (ie, is not specified or has a name the same as another module), the REST message will beep and say so.

4.11 Add to a model

This feature gives the user the option of adding any or all of the following items to a REST model:

1. Add a function declaration (to a module)

2. Add a variable declaration (to a module)

3. Add a *Connect* call (in the INIT section)

4. Add a *GetState* call (to a module)

To open the desired window to make additions, **Right** click and drag **Right** the *Add to a module...* button, and then release the button on the option preferred.

Only one of each of these windows will stay open at a given time. When the window is shown, provide all of the specifications which appear in boldface to add that item to the REST model.

To add the new item created, **Left** click the *Add this item* button. To hide the window, **Left** click the *Close this window* button. To iconify the window, **Left** click the boxed X in the upper left corner; iconifying is useful for bringing up a window without blanking out its attributes.

4.11.1 Add a function declaration

This option automatically inserts *Rate*, *Deathif*, *Pruneif*, and *Tranto* declarations and functions into specified module in the REST model.

To add a *Tranto* function, the *Rate* variable must already have been specified in the module.

Neither functions nor variables added may have the same name as a module, nor may they have the same name as any other identifier in their own module. Variables and functions of the same type are automatically grouped together.

A known deficiency of REST tool is the inability to allow the insertion of identifiers into a module where a word with the same name exists. For example, if the word 'tree' appears in a comment in a module, REST tool will not allow its name to be used with its automatic adding of a variable or function. The user must implement a manual override to make such an endeavor possible.

The Module name in the "Add to" dialogue window will be set, by default, to the module name in the main REST window.

4.11.2 Add a variable declaration

This option automatically inserts *State*, *Rate*, *Relation*, and *Generic Relation* variable declarations into the specified module in the REST model.

In most respects adding a variable declaration to a module is the same as adding a function declaration. One noteworthy feature should be highlighted: a new module skeleton will automatically be inserted [see §4.10] if a nonexistent module name is given in the *Relation* type (note that this will not be done for the *Generic* case).

With *State* and *Relation* variables, an array size of 1 indicates no array; it is the default. To increment or decrement the array size, **Left** click the appropriate arrow.

No module may be declared in its own *Relation*. (In addition, *Relations* do not need a message receipt function.)

4.11.3 Add a Connect call

When using this option, first position the caret in the text window by **Left** clicking the mouse at the exact place where the insertion is to occur.

Connect calls may only be inserted in the *Init* section of the REST file, and only with variables declared in the *Global* section. REST tool automatically checks for syntactic and semantic errors within the *Connect* call before inserting it into the REST model. The user must provide the explicit array connections after the *Connect* call is inserted.

An additional feature allows the *Connect* call to be nested in a for loop automatically if so desired. Also, by setting the *View* option, a *View* call can be set up to establish a one-way connection.

4.11.4 Add a GetState call

When using this option, first position the caret in the text window by **Left** clicking the mouse at the exact place where the insertion is to occur.

If no *Relation* is specified in the *GetState* window, the *GetState* is assumed to be working on the module in which the call will reside. REST tool automatically checks for syntactic and semantic errors within the *GetState* call before inserting it into the REST model. The user must provide the explicit array numbering after the *GetState* call is inserted.

A *GetNewState* call can be rendered instead of a *GetState* call simply by toggling the choice buttons in the window to the desired call.

Note that *Generic Relation GetState* calls must be produced manually.

4.12 Add a section

To use this feature, **Right** click on the button and drag the mouse to the right and down to the section desired. The following sections are available for addition: *Static*, *Global*, and *Init*. The section selected will be automatically inserted in the correct place in the REST model, if such a section does not already exist there.

4.13 Stuff selected text

To use this feature, **Left** click the caret in the text editor window [see §4.7] where the text will be inserted. Next, **Left** drag the mouse over the text, in some other window, to be

inserted. Finally, **Left** click the *Stuff selected text* button in the main panel to insert the selected text in the appropriate spot in the text editor window.

4.14 Goto in the text editor window

This feature moves the caret and user's view in the REST text window. **Left** click the button to jump to the module specified by the *Module name* line [see §4.9]. The view is also shifted to that point.

If the **Right** button is clicked and dragged right, a menu is pulled up which also provides the options of moving directly to the top or bottom of the text window. The view is also shifted to that point.

4.15 Set up a watch window

To create a new watch window, **Left** click the *Set up a watch window* button. Watch windows allow the user to observe several parts of the REST model at once.

As many watch windows can be opened at a time as desired, though it is recommended that the user open no more than can truly be watched at a time (to keep REST tool running fast). Module headings, functions, and the *Static*, *Global* and *Init* sections may be watched.

Watch windows are read-only, and they do not change as the text in the REST text window is being altered. To watch a module heading, the module name must be specified; to watch a function, the module name and function name must be specified.

To watch a selected item once in the watch window, **Left** click the *Watch* button. If the text is altered and then the watch button is pressed again, the contents of the watch window will be updated. To destroy the window, **Left** click the *Close this window* button. To iconify the window, **Left** click the boxed X in the upper left corner. Iconify the window only if fast access is a priority, as it is just as easy to open a new watch window.

The REST tool message area will report if there are mismatched braces, or if they are matched correctly, how many pairs exist in the current REST model. If the braces are mismatched in the REST model, the watch window will refuse to watch anything until the braces are matched.

If the *Module* option is selected and no module name is specified, the watch window will perform a **Watch All Modules**. It will list the names of all of the modules which are present in the current REST model, as well as the *Static*, *Global* and *Init* sections if they exist.

4.16 Open a term window

This feature opens a single terminal window for viewing the file system while REST tool is running. If more terminal windows are desired, use the command *xterm &* to create them. To iconify the terminal window, **Left** click the boxed X in the upper left hand corner of the window. To hide the terminal window, **Left** click the *Close this window* button.

When the window is first opened, the user is placed in the directory where the REST tool executable code resides.

The window may be resized by **Right** dragging the symbol in the upper right hand corner. In addition, the view may be split by **Right** dragging the mouse at the scrollbar. The view will be split in two at the place where the scrollbar was at the time of the split. To regain the caret in the terminal window, **Left** click the caret.

4.17 Run a REST model

When this main panel button is **Left** clicked, first the output window and then the run window pop up on the screen. Place them where you desire. *This feature will run the most recently saved version of the REST model, so be sure to save your creation before attempting a run.*

To run a REST model, first specify the model's file name, the desired mission time (an integer) and pruning level (a real number less than 1.0). An optional "report interval" that controls the frequency of reporting to the run-time display window may be specified. For a selected report interval of n, a processor will execute n path-records between each report update. Since report calculations do require computational resources, relatively infrequent intervals, e.g. 5000, are recommended. The file name will automatically be set to the current REST model's file name when the window is first opened. To select the machine on which the REST model will be run, **Right** click the little arrow and drag the mouse down to the remote machine name desired.

You can also specify the following values for **Detailed Event Count**: *all* , or a positive number n. This value controls the number of events that REST will record during a run. The default value is 100. REST records the n events with highest upper bound on their probability of occurrence These records are available after the run completes. Large values of n cause REST to run out of dynamic memory, a situation from which it does not exit gracefully.

The text file *termlist* should be altered to specify the machines on which REST can run. The first line of *termlist* must be **local**, which indicates that REST can be executed in the current directory. This line is then followed by 0 or more specifications of remote serial and

parallel machines using the following notation.

- Remote machine name *(like hilbert.cs.wm.edu)*
- Remote machine user name *(like nicol)*
- Remote REST directory *(like /mndisk/restuser/)*
- Remote directory to rcp the REST model *(like /mndisk/restuser/xfer/)*
- Type of remote machine *(either "serial" or "parallel")*

Use the *termlist* provided with this release of REST tool for emulation. This file should be altered prior to running the tool and provides the ability to run REST on a variety of serial and parallel machines.

The user can toggle back and forth between the various machines by using the *Machine to run REST model* menu. If the REST model has already been run once on a machine, and no changes have been made to the source since then, **Left** clicking the *Rerun model* button will run the model on the machine, with or without new parameters, without having to go through the trouble of recompiling the problem specific code.

The user can also select whether the run should be "smart", or "fast"; with the former selection being default. "Smart" runs perform array bounds checking on all references to state variables, and in debug mode provide more information about the model behavior. The "Fast" selection should be made only on production runs; it has been observed to deliver as much as a factor of two speedup over the standard method.

Left clicking the *Show outputs* button will display the output windows (display, debug and state records) without doing a REST model run. The *Quit* button is used to terminate the run tool presently; at most one run tool can be open at a time.

Left clicking the *Run model* button will attempt to run the specified REST model. This usually takes a while, so the cursor is changed to the busy signal to let the user know that the model is running and not hanging. To stop the model presently running, **Left** click the *Kill run* button.

Left clicking the *Report* button after a run will print out event files generated by the most recent run. Detailed Events give path sequences describing the events; SummaryByArray events are aggregations obtained by ignoring specific module variable array indices; SummaryByType are aggregations obtained by ignoring array indices, and further aggregating based on module type. The events are listed in decreasing order of probability, and the "cum" field for an event gives the sum of probabilities of that event and all events ahead of it in the list. The "cum" field of the last event thus indicates the significance of the reported events relative to the sum of all death-state probabilities.

Turning on the *Debug* option allows the user to debug the REST model run. For a list of commands, type "HELP" at the command line in the debug window. Only serial models may be debugged.

To provide information during the simulation a statistics window is attached to the bottom of the output window. The statistics are periodically updated. Typical values for the report frequency range from 1000 (often) to 100000 (infrequent). However, if no such report is desired (to speed up the simulation), set the report frequency range to 0. The following vital statistics are shown in the statistics window:

- Estimated completed work
- Average processor utilization
- Number of processors
- Number of paths processed
- Number of nodes generated
- Number of paths pruned
- Upper bound on pruning
- Lower bound on pruning
- Sum of probabilities of pruned paths
- Instantaneous pruning bound
- Number of death states
- Upper bound on death state probability
- Lower bound on death state probability

As is the case with the other subwindows, **Left** clicking the *Close this window* button will hide the window, while **Left** clicking the boxed X in the upper left hand corner will iconify the window.

When a run has been successfully completed, the run display window will contain the output of the execution shown. Use the scrollbar and mouse to view the window. The text in the window is read-only. User options in this window are achieved by **Left** clicking the appropriate button:

Iconify window: the upper left boxed X.

Close window: the *Close* button.

Clear template: the *Clear* button.

Save template: the *Save* button (file name must be specified).

To use the *run tool* without calling up REST tool, simply type at the shell prompt: '**xfilt** *REST model file name*'.

It should always be remembered that every invocation of REST generates files with fixed names, in a fixed portion of the file directory. Any "old" REST executables are thus automatically overwritten (except when the "rerun" option is selected). A user can save and later rerun old REST executions with the following sequence.

1. Generate the REST model.
2. Save the executable file obj/working/rest to another location. obj/working is wiped clean with each REST invocation.
3. To restore, move the executable back as obj/working/rest
4. Use the "rerun" option to re-execute.

4.17.1 Debugging

REST provides a debugging tool to aid the model development process. However, in order to use the tool effectively the modeler must understand the workings of REST analysis. We provide a sketch below.

The analysis employed by REST is based on a depth-first generation and analysis of the underlying state-space. The set of STATE variables from all instantiated module variables forms the state-vector for the state-space. Given a state-vector S, REST checks all DEATHIF conditions by testing each DEATHIF *Condition* function associated with each and every module variable. If S is not found to have failed, REST applies a pruning test. If the search is not pruned REST, generates all transitions possible from S, again by checking all TRANTO condition functions for all module variables. The set of transformed states (created by executing *Action* statements) are called the descendents of S. In classic depth-first fashion the list of descendents is attached to the front of a *working list*, and the first member of the list is selected for the same type of analysis and expansion as was performed on S. The size of the working list grows and shrinks as the computation progresses. The computation is known to have completed when the working list is empty.

This basic understanding is all that is necessary to debug a REST model. The Xrest Run window has a debug option, selectable by clicking the Debug "On" box. This option is selectable only when a serial execution engine is selected from termlist. Upon further selection of the Run model button (or Rerun model button, if appropriate) two new windows are brought up. The first, entitled "Debugger Tool", is the primary work area for debugging. The second, entitled "State Records", is used to display the entire REST model state vector.

The "command" line in the Debugger Tool is used to accept any one of a list of debugger commands, to be described below. A command is entered by typing it onto the command line and hitting keyboard's Return key. At any point a brief summary of legal commands can be called up by entering the command **help**. A longer summary is available with the command **HELP**.

Before discussing the debugger commands it is helpful to consider the overall structure of the debugger. Every RML model has two levels of breakpoints which are always automatically inserted. Calling the debugger allows those breakpoints to be enabled. Once at a breakpoint the debugger permits the examination of module state variables and the execution engine's working list. The order of states in the working list can be re-arranged at this time. Debugging options can also be enabled and disabled. At the coarsest level, a breakpoint exists just prior to the generation of a state's descendents, a so-called *Pre-generation breakpoint*. At a finer level, breakpoints can be enabled following each RML event.

REST encodes a state vector in terms the sequence of events applied to the initial state that result in the vector. This code appears within the debugger. The code is a sequence of pairs, where the first component of a pair is the identity of a module variable, and the second component is the identity of an event declared within such a module's MODULE definition. An example of such a description, taken from the debugger, is given below.

```
----- Pre-generation breakpoint:
Current path sequence
-----.(FTP[0],Fail).(PC[0],Crash).(FTP[0],2)
```

Here, `FTP[0]` and `PC[0]` are module variables (it is possible in this case that single module variables named `FTP` and `PC` are declared. This code treats all module variables as members of arrays). Recall that events can be symbolically labeled within a MODULE definition. That symbol is used as the second component in a pair, e.g., **Fail** and **Crash** above. Each event has the default symbol of its relative position in the MODULE declaration. Above, the symbol 2 means that the second event in the appropriate MODULE is the one being referenced. The sequence records the series of events, read left to right, that created the referenced state vector.

A description of the state vector (aka path sequence) about to be processed is given at every Pre-generation breakpoint. With pre-event breakpoints, the event just processed is

reported using this same code.

The following list describes legal debugger commands.

cont *(Continue)* Continue to the next breakpoint.

dcp *(display current path)* The code for the current path sequence is displayed. At an event breakpoint, the event just processed is included as part of that description.

dgl *(display generated paths)* A list of descendents of the last state vector processed are displayed. This should only be used at a Pre-generation breakpoint. It can be used at post-event breakpoints, but the effect is the same as **dwl** (see below).

dsv *(display state vector)* All STATE variables of all module variables are displayed, in the State Records window. This display is not updated automatically; **dsv** must be called every new time the state vector values are examined.

dwl *(display working list)* The entire working list is displayed.

exit Exit the debugger. This causes the debugger windows to disappear.

front The **dgl** and **dwl** commands list state vector in the order they appear in the working list. Command **front** with an integer argument i moves the state vector labeled i to the front of the list, thereby making it the next state vector to be processed.

hex Command **hex** + causes all subsequent displays of integer values to be given in hexadecimal. **hex** - turns this feature off.

log Command **log** + turns on a log of the the debugging session, **log** - turns it off. The log file name is rest.log, found in subdirectory obj/working. Be forewarned that executing **log** + causes any old version of rest.log to be lost.

prune This command allows the pruning level to be changed. It reports the current value, and prompts for a new one.

step Command **step** + enables breakpointing after every event. **step** - disables this feature.

trace Command **trace** + enables a detailed trace of REST analysis activity. Consider the segment below, obtained as part of execution for one event.

```
----- Breakpoint following TRANTO event FTP[0]:1
>> cont
Enter routine FTP[1]:GoodFTP
FTP[1]:GoodFTP:cond calls GetState(Status[0]) = 1
```

```
Enter routine FTP[1]:DeadFTP
FTP[1]:DeadFTP calls PutState(Status[0]), old = 1, new = 2
FTP[1]:DeadFTP sends message to (local)PrCont[0] = (global)PC[5]:LFTP[1]
Enter routine PC[5]:RecvFTPMsg
PC[5]:RecvFTPMsg:effect calls GetState(PControl[0]) = 0
FTP[1]:DeadFTP sends message to (local)PrCont[1] = (global)PC[6]:LFTP[2]
Enter routine PC[6]:RecvFTPMsg
```

First of all, in this case the step option has already been selected. A great deal of information is generation by the trace option, and its presentation is broken up into more manageable pieces under stepping. In the above trace, a TRANTO event has just been processed. The event is associated with module variable `FTP[0]`; the notation `:1` signifies event "1", in this case an automatic label. User supplied labels (e.g. `FailEvt` in Figure 2.1) are used here, when available. The trace reveals the processing of another TRANTO event. Recall that the processing of a state includes generation of all possible transformations. The trace shows the computation used to attempt one such transformation. A routine `GoodFTP()` is called to evaluate whether the status of module variable `FTP[1]` is GOOD. On the second line above we are told that routine `GetState()` is called to fetch the value of STATE variable `Status[0]`, and that the value returned is 1. Nomenclature `FTP[1]:GoodFTP:cond` identifies the module variable involved, the function involved, and the fact that the function is being called to support the conditional portion of the event. Apparently value 1 means the FTP is GOOD, because in the third line we observe a call to `DeadFTP()` which initiates the effects of its failure. In the fourth line we see that the variable `Status[0]` is modified to value 2 (FAILED); the fifth line reports the transmission of a message reporting the failure. The fragment

```
(local)PrCont[0] = (global)PC[5]:LFTP[1]
```

in that line indicates that the message is sent to the RELATION variable known locally as `PrCont[0]`. From a global perspective, that RELATION is bound to module variable `PC[5]`, and from `PC[5]`'s viewpoint the calling module is a RELATION module known to `PC[5]` as `LFTP[1]`. The sixth line shows that routine `RecvFTPMsg()`, bound to global module variable `PC[5]`, is called to handle the message. It, in turn, examines STATE variable `PControl[0]`. Message propagation apparently stops at that point, because the next line reports that control is reassociated with `FTP[1]`, who now sends a message to its RELATION module `PrCont[1]`, a message received by routine `RecvFTPMsg`, bound to module variable `PC[6]`. The event processing does not terminate here, this fragment is included to provide the user with the sense trace information.

trace - disables the trace option.

user user + enables the user control option. This option permits the user direct control over the next event to be applied to a state vector. Consider the sequence below.

```
----- Pre-generation breakpoint:
Current path sequence
-----.(FTP[0],1)
>> user +
Specify module array variable name.
>> NI
Specify module array variable (0-5).
>> 2
Specify event identifier.
>> 1
>> cont
----- Pre-generation breakpoint:
Current path sequence
-----.(FTP[0],1).(NI[2],1)
```

Entering user mode we are prompted to specify the array name of a declared module variable. We choose `NI`, and are prompted to specify which of the 6 members of that array we desire. Choosing `NI[2]`, we are then prompted for the identity of the event that should be executed. We choose the first event, and then instruct the system with a `cont` to execute the selected event. The next current path sequence shows that indeed the event we selected was executed, and that the resulting state vector is now at the head of the working list. The user option thus lets the user investigate specific event sequences, short-circuiting the automatic REST mechanisms for creating state vectors.

The system remains in user mode until the command **user -** is given to disable it. The system may also exit user mode if the user makes an error when prompted by the system. If there is ever any doubt,the **user +** can be reissued.

4.18 Module library

This feature provides the capabilities of saving a module to a library or inserting a precreated module from a library. The library is the directory whose name is specified by the *Library dir* line [see §4.9]. A blank line tells REST tool to use the current directory.

4.18.1 Insert from module library

To use the library features, **Right** click the mouse on the *Module library...* button, and **Right** drag the mouse until the library features menu appears. To insert a module, **Right** drag the mouse again and wait for the library modules available menu. Drag down the mouse to the module desired and release the **Right** mouse button. The module will be inserted at the beginning of the current REST model, provided no module with that name presently exists.

If there is a desire to insert modules not originally created and saved by REST tool, make sure the name of the file is *modulename.mod*, where *modulename* is the name of the module. Also make sure the file contains only the module to be inserted, because all contents of the file will be inserted at the beginning of the REST model.

4.18.2 Save to module library

To save a module to the library, make sure the module name and library directory name are specified in the appropriate lines [see §4.9]. REST tool will save the module in a file called *modulename.mod*, whose format is described above. Note that this action does not alter the current REST model.

4.19 Load a REST model

To load a REST model, first specify the file name in the *File name* line [see section §4.9], and then **Left** click the *Load a REST model* button. If the code has been modified since the last save, a notify window will pop up, requesting that the current file be saved first.

4.20 Save a REST model

To save a REST model, first specify the file name in the *File name* line [see section §4.9], and then **Left** click the *Save a REST model* button. Save will retain the previous save as a backup file in the same directory, with the file name appended by the extension BAK. **SAVE OFTEN!!!**

4.21 Clear the template

To clear the text window, **Left** click the *Clear the template* button. If the text was modified since the last save, a notify window will pop up asking which course of action is *REALLY*

desired. **If the template is cleared it cannot be retrieved** ('OOOPS!'), so plan the course of action carefully.

4.22 Quit REST tool

To quit from REST tool, **Left** click the *Quit REST tool* button. If the text was modified since the last save, a notify window will pop up asking which course of action is *REALLY* desired.

4.23 Program Enhancements

A variable or function name cannot take on as a name any word which appears in that identifier's module.

4.24 List of REST messages

The following messages may appear in the REST message area.

Addition subwindow closed The window has closed successfully. To use the window again, **Left** click the *Add to REST model* button in the main window.

Addition subwindow opened The window has opened without problem.

All modules listed The *Watch all modules* instruction was performed successfully [see §4.15], and all of the modules are listed in the watch window in the order they appear in the REST model.

Can only connect in Init section *Connect* calls can only be made in the Init section of the REST model, so the caret must be repositioned to a place within the Init section.

Clear aborted The clear attempt was unsuccessful, whether intended or not.

Cond and state funcs must be unique The names of the identifiers are the same, which is illegal in the REST world.

Connect added The *Connect* call has been added without problem.

Declaration in Global is bad A variable in the *Global* section has been declared of a module type which does not exist in the current REST model.

File not found No file of that name exists. Perhaps the path is not correct, or a capital letter is missing, or the file name has not been specified in the *File name* line, or something similar.

File not overwritten No save was attempted. The file name must be changed if a file save is desired to avoid overwritting the data file with the given file name.

Filter template cleared The filter template has been cleaned free of all outputs formerly contained within.

Filter template saved The contents of the filter template have been saved to the specified filename. Note that this will not clear the template.

Func name is not unique An identifier already exists with that name in that module.

Function added The function declaration in the module header and the accompanying function skeletons in the *Code* section were added to the module successfully.

Function not resident in module No function of that name has been declared in the module specified.

GetState added The GetState call has been added without a problem.

Global section does not exist The *Global* section of the REST model is necessary for the present instruction to execute.

Goto successful REST tool was able to reposition the caret.

Init section does not exist The *Init* section of the REST model is necessary for the present instruction to execute.

Library module was inserted No problems were encountered placing it at the beginning of the current REST model.

Load complete The REST model has been successfully loaded into REST tool.

Mod cannot be its own Relation In a *Relation* declaration, the module named as the *Relation* must be different from the name of the module into which this declaration will be placed.

Mod func and var names must be unique In a module header's *Relation* declaration, the message receipt function and variable name must be different from each other and every other identifier in the module. An exception: different *Relations* can share the

same message receipt function, in which case a REST message will inform the user that such a thing has been done.

Module already exists The module name chosen is already in this REST model.

Module does not exist The module name chosen is not in this REST model.

Module library is empty No files ending in *.mod* reside in the specified library directory.

Module not saved to library The save was aborted successfully.

Module saved to library A file with the name *modulename.mod* has been created successfully in the specified library directory.

Msg rec func exists; new Mod added This is actually two messages in one. See the messages for *Msg rec func name not unique* and *Variable and new Module added.*

Msg rec func name not unique A function with that name already exists in that module; REST tool assumes that the duplicate name was intended, for example when using the *Null()* function.

Must specify a pruning level The REST model cannot be run without a numeric pruning level given.

Must specify module Var name During the process of creating a *GetState* call, the name of the module name has been specified; the variable of that type declared within the module must be used instead.

Must use Relation variable In creating a *Connect* call, the module name has been specified instead of the variable of that type declared in the *Global* section of the REST model.

New module created The module was successfully added to the beginning of the REST model.

No file name selected A file name must be specified in the main window's *File name* line before a REST model can be saved.

No function name given A function name must be specified before REST tool can execute the instruction.

No module name selected A module name must be given before REST tool can execute the instruction.

No modules exist No modules have been created yet in the current REST model.

No module was inserted The *Add a library module* was aborted.

No variable name given A variable name must be specified before REST tool can execute the instruction.

Quit aborted The quit attempt was unsuccessful, whether intended or not.

Rate variable does not exist To add a *Tranto* function, a *Rate* variable of that name must already have been defined in that module. To add the function, then, call up the *Add a variable* window, add the *Rate* variable, and then **Left** click the *Add this function* button in the *Add a function* window.

Relation module not declared The *GetState* module specified does not presently reside in the current REST model.

Relation module is generic The *GetState* module specified has been declared of the generic type. The GetState call must be entered manually to ensure that the user gets what the user wants.

Relation not declared in Global The *Connect* call can only connect two modules whose variables have been declared in the *Global* section.

Relation not declared in module The *Connect* call can only connect two modules whose variables have been declared in the proper modules.

REST model run complete The REST model has finished running, with its output appearing in the filter display window.

REST model running The REST model is running, but this has been known to take a while, so be patient.

Run window opened The *Run a REST model* window has been opened without problem.

Save failed The save command was not successful.

Save succeeded The save command was successful. REST tool can now be cleared the template or exited. When a REST model is saved, the last version is also saved in the name *filename* appended by a % sign, in the same directory that the REST model is saved.

Section added The section has been added successfully to the proper position in the REST model.

Section does not exist The section referenced (either *Static*, *Global* or *Init*) does not exist presently in the REST model.

Some var attributes not given One or more variable attributes has been left blank, and they must be specified before REST tool can execute the instruction.

State var not declared in module When creating a *GetState* call, the state to be retreived does not exist in the module specified.

Template cleared The main window's text template has been successfully wiped clean. Useful in building another REST model without exiting REST tool.

Template modified; please save or clear A new REST model cannot be loaded into the text editor window until the current REST model has been saved or cleared since its last modification.

Term window opened The terminal window has been opened without problem.

There are ? pairs of {}'s Gives a count of the number of braces pairs in the REST model. If this message is displayed, the watch instruction has executed correctly, and the watch window now is looking at the intended item.

There are too many braces In order for a watch window to successfully watch an item, the number of open braces in the REST model must equal the number of closed braces. This message alerts the user to an imbalance which must be remedied before a watch can be performed.

Think again A REST model must have something in it before it can be saved.

This feature not yet available The current release of REST tool does not support this feature.

Var name is not unique An identifier already exists with that name in the module.

Variable added The variable declaration and accompanying function skeletons have been added to the module successfully.

Variable and new Module added The variable declaration and function skeleton for the Relation have been added, and the module named in the declaration previously did not exist. It has been created and inserted at the beginning of the REST model.

Watch window added A new watch window has been created successfully.

Chapter 5

Listings

```
Comment = 0;
"Echo = 0;"
"prune = 1.0E-14;"
PFail = 1.0E-4;

space =  (P: array[1..4] of 0..1);
Start = (4 of 0);

deathif(P[1]+P[2]+P[3]+P[4]>1);

for i=1,4;
 if (P[i]=0) tranto P[i]=1 by PFail;
endfor;
```

Listing 1. ASSIST Model of Non-Reconfiguring Quad

```
MODULE Processor {
   STATE ProcState;
   RATE ProcFail;

   IF ProcGood() TRANTO FailEffect() BY ProcFail;

   CODE {

//FAILEFFECT
         void FailEffect() {
            PutState(ProcState,FAIL);
            } //FailEffect

//PROCGOOD
         int ProcGood() {
            return(ISGOOD(GetState(ProcState)));
            } //ProcGood

        } //CODE
    } //Processor

 MODULE System {
     RELATION Processor  Procs[4];

     DEATHIF DeathCond();

     CODE {

//  Death if 2 or more failed processors
        int DeathCond() {
           int d,i;
           d=0;
           for (i=0; i<NumProcs; i++)
              if (FAILED(GetState(Procs[i].ProcState))) d++;
           return(d>1);
           } // DeathCond

        } //CODE

     } //System
```

Listing 2a. RML Model of Non-Reconfiguring Quad

```
STATIC
{
#define GOOD 1
#define FAIL 2
#define NumProcs 4
#define ISGOOD(s) (s&GOOD)
#define FAILED(s) (s&FAIL)
}

GLOBAL {
   Processor: P[4];
   System: SYS;
   }

INIT

   {
   int i;
   for (i=0; i<NumProcs; i++)
     {
     View(SYS.Procs[i],P[i]);
     Set(P[i].ProcState,GOOD);
     }
   Rate(Processor.ProcFail,1.0E-4);
   }
```

Listing 2b. RML Model of Non-Reconfiguring Quad (contd)

```
comment = 0;
"Echo = 0;"
"prune=1.0E-14;"
PFail = 1.0E-4;
Prec = 3600;

   space = (Pgood :array[1..4] of 0..1, Pbad: array[1..4] of 0..1);


   Start = (4 of 1, 4 of 0);


   deathif ((Pbad[1]+Pbad[2]+Pbad[3]+Pbad[4])>=
               (Pgood[1]+Pgood[2]+Pgood[3]+Pgood[4]));

for i=1,4;
   if (Pgood[i]=1) tranto Pgood[i]=0,Pbad[i]=1 by PFail;

   if (Pbad[i]=1) tranto Pbad[i]=0 by FAST Prec;
endfor;
```

Listing 3. ASSIST Model of Reconfiguring Quad

```
MODULE Processor {
   STATE ProcState;
   RATE ProcFail;
   RELATION FTP FTP: FTPmess();

Processor: IF ProcGood() TRANTO FailEffect() BY ProcFail;

   CODE {

//FAILEFFECT
        void FailEffect() {
            PutState(ProcState,FAIL);
  //    Notify FTP Parent that a processor has failed
            SendValue(FTP,FAIL);
            } //FailEffect

//PROCGOOD
         int ProcGood() {
            return(GetState(ProcState)&GOOD);
            } //ProcGood

//FTPMESS
   void FTPmess(msg,who)
        int *msg, who; {
//  FTP can alter child's state.  Typically to REMOVE from working set
      switch (*msg) {
         case REMOVED: PutState(ProcState,REMOVED); break;
         } //switch
      } //FTPmess
   } //CODE
} //Processor
```

Listing 4a. RML Model of Reconfiguring Quad: Processor Module

```
 MODULE FTP {
     STATE FTPState;
     RATE FTPrec;
     RELATION Processor  P[4]: Pmess();

FTP: IF Vulnerable() TRANTO Recover() BY FAST FTPrec;

     CODE {
//EVAL
//  This function counts GOOD and BAD processors (Maybe I should call
//  it SantaClaus) and sets FTPState GOOD, VULNERABLE or BAD accordingly
        void Eval() {
           int g,b,p;
           g=b=0;
           for (p=0; p<NumProcs; p++)  {
              if (GetNewState(P[p].ProcState)==GOOD) g++;
              else if (GetNewState(P[p].ProcState)==FAIL) b++;
              }
           if (b==0) PutState(FTPState,GOOD);
           if ( (b>=g)||(g==0) ) PutState(FTPState,FAIL);
           } //Eval

//VULNERABLE
         int Vulnerable() {
            int p,f;
            f=0;
            for (p=0; p<NumProcs; p++)
              if (GetState(P[p].ProcState)==FAIL) f++

            return(f>0);
            } //Vulnerable
```

Listing 4b. RML Model of Reconfiguring Quad: FTP Module

```
//RECOVER
//  If a Proc is BAD we REMOVE it from WORKING set
         void Recover() {
            int p;
            for (p=0; p<NumProcs; p++)
              if (GetState(P[p].ProcState)==FAIL) SendValue(P[p],REMOVED);

            FTP_Eval();
         } //Recover

//Pmess
//  A Proc has changed state, so we should re-evaluate FTP
       void Pmess(msg,who)
            int *msg,who; {
           FTP_Eval();
          } //Pmess
      } //CODE
}// FTP
```

Listing 4c. RML Model of Reconfiguring Quad: FTP Module (contd)

```
MODULE System {

     RELATION FTP FTP;

     DEATHIF FTPFail();

     CODE
        {

//FTPFAIL
        int FTPFail() {
           return(GetState(FTP.FTPState)==FAIL);
           } FTPFail

       } //CODE

}//System
```

Listing 4d. RML Model of Reconfiguring Quad: System Module

```
STATIC
{
#define GOOD 1
#define FAIL 2
#define REMOVED 4
#define NumProcs 4
}

GLOBAL {
   Processor: P[4];
   FTP: FTP;
   System: SYS;
   }


INIT
   {
   int p;
   for (p=0; p<NumProcs; p++)
     {
     Connect(FTP.P[p],P[p].FTP);
     Set(P[p].ProcState,GOOD);
     }
   Set(FTP.FTPState,GOOD);
   View(SYS.FTP,FTP);
   Rate(FTP.FTPrec,3600.0);
   Rate(Processor.ProcFail,1.0E-4);
   }
```

Listing 4e. RML Model of Reconfiguring Quad: Initialization

```
MODULE Processor {
   STATE ProcState;
   RATE ProcFail;

//  Relation with sibling voter
   RELATION Voter VME: Vmess();

   RELATION FTP FTP;

   IF (GetState(ProcState)&GOOD) TRANTO FailEffect() BY ProcFail;

   CODE {

//FAILEFFECT
         void FailEffect() {
            int p,v;
            PutState(ProcState,FAIL);

            SendValue(VME,ERROR);

//  Tell Parent FTP we've failed (Voter could do this as error report
//  But then all 4 voters would send message
            SendValue(FTP,FAIL);
            } //FailEffect

//VMESS
//  When Voter Fails the sibling Processor Module also fails
         void  Vmess(msg,who)  {
              Processor_FailEffect();
         } //Vmess
     } //CODE
} //Processor
```

Listing 5a. RML Model of Quad with Voter: Processor Module

```
 MODULE Voter {
    STATE VoterState,VoterErrors,VoterEnable;
// Relation with sibling processor
    RELATION Processor PME: Pmess();

//  Transmit Ports
    RELATION Voter VSend[4];
//  Receive Ports
    RELATION Voter VReceive[4]: Vmess();

    RELATION FTP FTP: FTPmess();


    CODE {

// FAILEFFECT
// When Voter Fails it notifies its sibling processor
       void FailEffect() {
          PutState(VoterState,FAIL);
          SendValue(PME,FAIL);
          } //FailEffect

//EVAL
      void Eval() {
         int g,b,e,v,p;
         g=b=0;
         e = GetNewState(VoterErrors);
         v = GetNewState(VoterEnable);
         e=v&e;   /*  e will hold bit vector of BAD ENABLED processors */
         v=v^e;   /* If ENABLEd processors aren't BAD, they're GOOD */
// Count errors and see if majority rules
         for (p=0; p<=3; p++) {
           if (e&(1<<p)) b++;
           if (v&(1<<p)) g++;
           }
         PutState(VoterState,GOOD);
         if (b>0) PutState(VoterState,ERROR);
         if ((b>=g)||(g==0)) Voter_FailEffect();
      } //Eval
```

Listing 5b. RML Model of Quad with Voter: Voter Module

```
//PMESS
      void Pmess(msg,who)
        int *msg,who; {
         int v;
         switch (*msg) {
           case FAIL:
           case ERROR: if (!(GetNewState(VoterState)&FAIL))
                              for (v=0; v<4; v++) SendValue(VSend[v],ERROR);
           } //switch
        } //Pmess

//VMESS
//  Voter sends ERROR message when it's Processor fails
      void Vmess(msg,who)
         int *msg,who; {
           int t;
// The Voter ID's go from 1,2,4,8, but the indices go from 0..3
           switch (*msg) {
             case ERROR:
                t = GetNewState(VoterErrors)|((1<<who)&
                     GetNewState(VoterEnable));
                PutState(VoterErrors,t);
                Voter_Eval();
                } //switch
           } //Vmess

//FTPMESS
//  FTP will reset enable register when it recovers
      void FTPmess(msg,who)
          int *msg,who; {
          int e;
          PutState(VoterEnable,*msg);
//  Disabled Channels are no longer Bad
          e = GetState(VoterErrors);
          e = e&(*msg);
          PutState(VoterErrors,e);
          Voter_Eval();
          } //FTPmess
      }//CODE
}//Voter
```

Listing 5c. RML Model of Quad with Voter: Voter Module (contd)

```
 MODULE FTP {
     STATE FTPState,WorkingSet;
     RATE FTPrec;
     RELATION Voter  V[4];
     RELATION Processor P[4]: Pmess();

     IF Vulnerable() TRANTO Recover() BY FAST FTPrec;

     CODE {

//EVAL
        void Eval() {
           int g,b,p;
           g=b=0;
// Poll Processors and Voters.  FTP is OK as long as it has ANY good
processors
// If any voters have errors FTP will start recovery
           for (p=0; p<NumProcs; p++) {
             if (GetNewState(P[p].ProcState)==GOOD) g++;
             }
           PutState(FTPState,GOOD);
           if (g==0) PutState(FTPState,FAIL);
          } //Eval

//PMESS
// A processor has changed state so re-eval FTP
         void Pmess(msg,who)
            int *msg,who; {
            FTP_Eval();
            }//Pmess

//VULNERABLE
         int Vulnerable() {
           int p,b;
           b=0;
           for (p=0; p<NumProcs; p++; {
             if ( ((1<<p)&GetState(WorkingSet)) &&
              (GetState(V[p].VoterState)==ERROR) ) b++;
             }
           return(b>0);
           }
```

Listing 5d. RML Model of Quad with Voter: FTP Module

```
//COUNTS
// There may be a better way of doing this
// This function accumulates error count for each processor
        void Count(s,p,P1e,P2e,P3e,P4e)
           int s,p,*P1e,*P2e,*P3e,*P4e; {
           if (p&GetState(WorkingSet)) {
              if (P1ID&s) *P1e = *P1e + 1;
              if (P2ID&s) *P2e = *P2e + 1;
              if (P3ID&s) *P3e = *P3e + 1;
              if (P4ID&s) *P4e = *P4e + 1;
              }
           }//Counts

//RECOVER
        void Recover() {
           int e,s,P1e,P2e,P3e,P4e,v;
           e=P1e=P2e=P3e=P4e=0;

// For each voter, count errors for each processor
           for (v=0; v<NumProcs; v++)
              if ((1<<v)&GetState(WorkingSet)) {
                 s = GetState(V[v].VoterErrors);
                 FTP_Count(s,(1<<v),&P1e,&P2e,&P3e,&P4e);
                 }
// If more than 2 voters agree that a Proc has errors, then we believe it
           if (P1e>1) e = e|P1ID;
           if (P2e>1) e = e|P2ID;
           if (P3e>1) e = e|P3ID;
           if (P4e>1) e = e|P4ID;
// Remove Bad Procs from working set
           e = GetState(WorkingSet)&(~e);
           PutState(WorkingSet,e);
// Reset Voter enable latches
           for (v=0; v<NumProcs; v++)
               if (e) SendValue(V[v],e);
// Re-eval FTP
   FTP_Eval();
   }//Recover
      }//CODE
}FTP
```

Listing 5e. RML Model of Quad with Voter: FTP Module (contd)

```
MODULE System {

     RELATION FTP FTP;

     DEATHIF FTPFail();

     CODE   {

//FTPFAIL
        int FTPFail()  {
           return(GetState(FTP.FTPState)==FAIL);
           }//FTPFail

      }//CODE
}//System


STATIC  {
#define GOOD 1
#define FAIL 2
#define REMOVED 4
#define VULNERABLE 8
#define ERROR 16
#define P1ID 1
#define P2ID 2
#define P3ID 4
#define P4ID 8
#define NumProcs 4
}
```

Listing 5f. RML Model of Quad with Voter: System Module and Static Section

```
GLOBAL {
   Processor: P[4];
   Voter: V[4];
   FTP: FTP;
   System: SYS;
   }

INIT {
   int i,j;
   for (i=0; i<NumProcs; i++) {
     Connect(FTP.V[i],V[i].FTP);
     Connect(V[i].PME,P[i].VME);
     for (j=0; j<NumProcs; j++)
         Connect(V[i].VSend[j],V[j].VReceive[i]);
     Connect(FTP.P[i],P[i].FTP);
    }

    View(SYS.FTP,FTP);

    for (i=0; i<NumProcs; i++) {
      Set(P[i].ProcState,GOOD);
      Set(V[i].VoterState,GOOD);
      Set(V[i].VoterErrors,0);
      Set(V[i].VoterEnable,(P1ID|P2ID|P3ID|P4ID));
     }

   Set(FTP.FTPState,GOOD);
   Set(FTP.WorkingSet,(P1ID|P2ID|P3ID|P4ID));
   Rate(FTP.FTPrec,3600.0);
   Rate(Processor.ProcFail,1.0E-4);
}
```

Listing 5g. RML Model of Quad with Voter: Initialization

```
Comment = 0;
"Echo = 0;"
"prune=1.0E-14;"
IFail = 1.0E-5;
Irec = 3600;

   space = (Igood: array[1..4] of 0..1, IO: array[1..4] of 0..1,
            IOError: array[1..4] of 0..1);


   Start = (4 of 1, 1, 0, 0, 1, 4 of 0);


   deathif (IO[1]+IO[2]+IO[3]+IO[4]=0);


for i=1,4;
   if (Igood[i]=1) and (IO[i]=1) tranto Igood[i]=0, IO[i]=0, IOError[i]=1 by
IFail;
   if (Igood[i]=1) and (IO[i]=0) tranto Igood[i]=0 by IFail;

endfor;

   if (IOError[1]=1) and
      (Igood[2]=1) tranto IOError[1]=0, IO[2]=1 by FAST Irec;
   if (IOError[1]=1) and
      (Igood[2]=0) tranto IOError[1]=0 by FAST Irec;
   if (IOError[2]=1) tranto IOError[2]=0 by FAST Irec;
   if (IOError[3]=1) tranto IOError[3]=0 by FAST Irec;
   if (IOError[4]=1) and
      (Igood[3]=1) tranto IOError[4]=0, IO[3]=1 by FAST Irec;
   if (IOError[4]=1) and
      (Igood[3]=0) tranto IOError[4]=0 by FAST Irec;
```

Listing 6. ASSIST model of Interface

```
  MODULE Interface {
    STATE IntState;
    RATE IntRate;

    RELATION InterfaceManager IntMan: Manmess();

// Not use of '&' instead of '=='. This is because an Interface can be either
// GOOD or (GOOD|INUSE). In either case it can fail. The difference is the
// effect of failing if INUSE or not. Also note that for this change the value
// of GOOD is made 1 and other State values modified accordingly.

    IF (GetState(IntState)&GOOD) TRANTO FailEffect() BY IntRate;

    CODE {

// FAILEFFECT
//  When an Interface fails it produces errors
       void FailEffect() {
          int s;
//  Have to retain INUSE bit
          s = GetState(IntState);
          PutState(IntState,((s&(~GOOD))|FAIL|ERROR));
          SendValue(IntMan, FAIL);
          }//FailEffect

//MANMESS
//  The Interface's Redundancy Manager can place the Interface Either
INUSE
//  or not INUSE
       void Manmess(msg,who)
          int *msg,who; {
          int s;
          s=GetState(IntState);
          switch (*msg) {
            case INUSE: PutState(IntState,s|INUSE); break;
            case REMOVED: PutState(IntState,s&(~INUSE)); break;
            } //switch
          }//Manmess
       }//CODE
 }Interface
```

Listing 7a. RML Model of Interface: Interface Module

```
MODULE InterfaceManager {
   STATE IntManState;
   RATE IntManRec;
   RELATION Interface I[4]: Imess();

   IF (GetState(IntManState)==VULNERABLE) TRANTO
       RecEffect() BY FAST IntManRec;

   CODE {

//EVAL
// To evaluate Health of I/O system, only consider active (INUSE) elements
        void Eval() {
           int g,b,s,i;
           g=b=0;
           for (i=0; i<4; i++) {
              s = GetNewState(I[i].IntState);
              if (s&INUSE)
                if (s&ERROR)
                  b++;
                else
                  g++;
             }

           PutState(IntManState,GOOD);
           if (b>0) PutState(IntManState,VULNERABLE);
           if (g==0) PutState(IntManState,FAIL);
           } //Eval

// IMESS
//  When an Interface changes state, The parent manager must reevaluate
        void Imess(msg,who)
           int *msg,who; {
            InterfaceManager_Eval();
           } //Imess
```

Listing 7b. RML Model of Interface: InterfaceManager Module

```
//  RECEFFECT
//  This recovery is pretty simple.
//  Interfaces which are INUSE and have ERRORS are REMOVED.
//  If I[0] fails, I[1] is activated and if I[3] fails, I[2] is activated
        void RecEffect(){
           int s,i;
           for (i=0; i<NumProcs; i++) {
              s = GetState(I[i].IntState);
              if ((s&INUSE)&&(s&ERROR)) {
                SendValue(I[i],REMOVED);
                switch (i) {
                  case 0: if (GetNewState(I[1].IntState)&GOOD)
                                        SendValue(I[1],INUSE); break;
                  case 3: if (GetNewState(I[2].IntState)&GOOD)
                                        SendValue(I[2],INUSE); break;
                  } //switch
                }//if
              }//for

// Re-eval the Interface
           InterfaceManager_Eval();
           }//RecEffect
       }//CODE
 } //InterfaceManager
```

Listing 7c. RML Model of Interface: InterfaceManager Module (contd)

```
MODULE System {

     RELATION InterfaceManager IntMan;

     DEATHIF INTFail();

     CODE {

//INTFAIL
        int INTFail() {
           return(GetState(IntMan.IntManState)==FAIL);
           }//INTFail
        }//CODE
}//System

STATIC  {
#define GOOD 1
#define FAIL 2
#define REMOVED 4
#define VULNERABLE 8
#define ERROR 16
#define INUSE 32
#define P1ID 1
#define P2ID 2
#define P3ID 4
#define P4ID 8
#define NumProcs 4
}
```

Listing 7d. RML Model of Interface: System Module and Static Section

```
GLOBAL {
   Interface: Int[4];
   InterfaceManager: IntMan;
   System: SYS;
   }

INIT {
   int i,j;
   for (i=0; i<NumProcs; i++) {
     Connect(IntMan.I[i], Int[i].IntMan);
     }

    View(SYS.IntMan,IntMan);

   Set(Int[0].IntState,(GOOD|INUSE));
   Set(Int[1].IntState,GOOD);
   Set(Int[2].IntState,GOOD);
   Set(Int[3].IntState,(GOOD|INUSE));

   Set(IntMan.IntManState,GOOD);
   Rate(Interface.IntRate,1.0E-5);
   Rate(InterfaceManager.IntManRec,3600.0);
   }
```

Listing 7e. RML Model of Interface: Initialization

```
Comment = 0;
"Echo = 0;"
"prune=1.0E-14;"
PFail = 1.0E-4;
Prec = 3600;
IFail = 1.0E-5;
Irec = 3600;

   Space = (Pgood :array[1..4] of 0..1, Pbad: array[1..4] of 0..1,
               Igood: array[1..4] of 0..1,  IO: array[1..4] of 0..1,
               IOError: array[1..4] of 0..1);

   Start = (4 of 1, 4 of 0, 4 of 1, 1, 0, 0, 1, 4 of 0);

   deathif ((Pbad[1]+Pbad[2]+Pbad[3]+Pbad[4])>=
             (Pgood[1]+Pgood[2]+Pgood[3]+Pgood[4]));
   deathif (IO[1]+IO[2]+IO[3]+IO[4]=0);

for i=1,4;
   if (Pgood[i]=1) and (Igood[i]=1) and (IO[i]=1) tranto
       Pgood[i]=0, Pbad[i]=1, Igood[i]=0, IO[i]=0, IOError[i]=1 by PFail;
   if (Pgood[i]=1) and (Igood[i]=1) and (IO[i]=0) tranto
       Pgood[i]=0, Pbad[i]=1, Igood[i]=0 by PFail;
   if (Pgood[i]=1) and (Igood[i]=0) and (IO[i]=0) tranto
       Pgood[i]=0, Pbad[i]=1 by PFail;
   if (Pgood[i]=1) and (Igood[i]=0) and (IO[i]=1) tranto
      Pgood[i]=0 by PFail;
   if (Pbad[i]=1) tranto Pbad[i]=0 by FAST Prec;

   if (Igood[i]=1) and (IO[i]=1) tranto Igood[i]=0, IO[i]=0, IOError[i]=1 by
IFail;
   if (Igood[i]=1) and (IO[i]=0) tranto Igood[i]=0 by IFail;
endfor;

   if (IOError[1]=1) and (Igood[2]=1) tranto
       IOError[1]=0, IO[2]=1 by FAST Irec;
   if (IOError[1]=1) and (Igood[2]=0) tranto IOError[1]=0 by FAST Irec;
   if (IOError[2]=1) tranto IOError[2]=0 by FAST Irec;
   if (IOError[3]=1) tranto IOError[3]=0 by FAST Irec;
   if (IOError[4]=1) and (Igood[3]=1) tranto
       IOError[4]=0, IO[3]=1 by FAST Irec;
  if (IOError[4]=1) and (Igood[3]=0) tranto IOError[4]=0 by FAST Irec;
```

Listing 8. ASSIST Module of Quad Processors plus Intefaces

```
MODULE Processor {
  STATE ProcState;
  RATE ProcFail;

//  Relation with sibling voter
  RELATION Voter VME: Vmess();

  RELATION FTPmodule FTP;

  RELATION Interface Int;

  IF (GetState(ProcState)&GOOD) TRANTO FailEffect() BY ProcFail;

  CODE {

//FAILEFFECT
        void FailEffect() {
           int p,v;
           PutState(ProcState,FAIL);

            SendValue(VME,ERROR);

//  Tell Parent FTP we've failed (Voter could do this as error report
//  But then all 4 voters would send message
           SendValue(FTP,FAIL);
           SendValue(Int,FAIL);
           }//FailEffect

// VMESS
//  When Voter Fails the sibling Processor Module also fails
     void  Vmess(msg,who)
        int *msg, who; {
           Processor_FailEffect();
        } //Vmess
        }//CODE
    }//Processor
```

Listing 9a. RML Model of Quad Processors plus Interfaces: Processor Module

```
 MODULE Voter {
    STATE VoterState,VoterErrors,VoterEnable;

// Relation with sibling processor
    RELATION Processor PME: Pmess();

//  Transmit Ports
    RELATION Voter VSend[4];
//  Receive Ports
    RELATION Voter VReceive[4]: Vmess();

    RELATION FTPmodule FTP: FTPmess();

    CODE {

//FAILEFFECT
// When Voter Fails it notifies its sibling processor
       void FailEffect() {
          PutState(VoterState,FAIL);
          SendValue(PME,FAIL);
          }//FailEffect

//EVAL
      void Eval() {
         int g,b,e,v,p;
         g=b=0;
         e = GetNewState(VoterErrors);
         v = GetNewState(VoterEnable);
         e=v&e;    /*  e will hold bit vector of BAD ENABLED processors */
         v=v^e;    /* If ENABLEd processors aren't BAD, they're GOOD */
// Count errors and see if majority rules
         for (p=0; p<=3; p++) {
           if (e&(1<<p)) b++;
           if (v&(1<<p)) g++;
           }
         PutState(VoterState,GOOD);
         if (b>0) PutState(VoterState,ERROR);
         if ((b>=g)||(g==0)) Voter_FailEffect();
       }//Eval
```

Listing 9b. RML Model of Quad Processors plus Interfaces: Voter Module

```
//PMESS
      void Pmess(msg,who) {
         int v;
         if (!(GetNewState(VoterState)&FAIL))
             for (v=0; v<4; v++) SendValue(VSend[v],ERROR);
        } //Pmess

//VMESS
//  Voter sends ERROR message when it's Processor fails
      void Vmess(msg,who)
         int *msg,who; {
           int t;
// The Voter ID's go from 1,2,4,8, but the indices go from 0..3
           t =
GetNewState(VoterErrors)|((1<<who)&GetNewState(VoterEnable));
           PutState(VoterErrors,t);
           Voter_Eval();
           }//Vmess

//FTPmess
//  FTP will reset enable register when it recovers
       void FTPmess(msg,who)
           int *msg,who; {
           int e;
           PutState(VoterEnable,*msg);
//  Disabled Channels are no longer Bad
           e = GetState(VoterErrors);
           e = e&(*msg);

           PutState(VoterErrors,e);

           Voter_Eval();

           }//FTPmess
        }//CODE
 }//Voter
```

Listing 9c. RML Model of Quad Processors plus Interfaces: Voter (contd)

```
MODULE FTPmodule {
     STATE FTPState,WorkingSet;
     RATE FTPrec;
     RELATION Voter  V[4];
     RELATION Processor P[4]: Pmess();

     IF Vulnerable() TRANTO Recover() BY FAST FTPrec;

     CODE {

//EVAL
        void Eval()
           {
           int g,b,p;
           g=b=0;
// Poll Processors and Voters.  FTP is OK as long as it has ANY good
processors
// If any voters have errors FTP will start recovery
           for (p=0; p<NumProcs; p++)
              if (GetNewState(P[p].ProcState)==GOOD) g++;

           PutState(FTPState,GOOD);
           if (g==0) PutState(FTPState,FAIL);
          } //Eval

//PMESS
// A processor has changed state so re-eval FTP
         void Pmess(msg,who)
            int *msg,who; {
            FTPmodule_Eval();
            }//Pmess

//VULNERABLE
         int Vulnerable() {
           int p,b;
           b=0;
           for (p=0; p<NumProcs; p++)
               if ( ((1<<p)&GetState(WorkingSet)) &&
              (GetState(V[p].VoterState)==ERROR) ) b++;
           return(b>0);
           }//Vulnerable
```

Listing 9d. RML Model of Quad Processors plus Interfaces: FTP Module

```
//COUNTS
// There may be a better way of doing this
// This function accumulates error count for each processor
         void Count(s,p,P1e,P2e,P3e,P4e)
            int s,p,*P1e,*P2e,*P3e,*P4e;{
            if (p&GetState(WorkingSet)) {
               if (P1ID&s) *P1e = *P1e + 1;
               if (P2ID&s) *P2e = *P2e + 1;
               if (P3ID&s) *P3e = *P3e + 1;
               if (P4ID&s) *P4e = *P4e + 1;
               }
            }//Counts

//RECOVER
         void Recover() {
            int e,s,P1e,P2e,P3e,P4e,v;
            e=P1e=P2e=P3e=P4e=0;

// For each voter, count errors for each processor
            for (v=0; v<NumProcs; v++) {
               s = GetState(V[v].VoterErrors);
               FTPmodule_Count(s,(1<<v),&P1e,&P2e,&P3e,&P4e);
               }
// If more than 2 voters agree that a Proc has errors, then we believe it
            if (P1e>1) e = e|P1ID;
            if (P2e>1) e = e|P2ID;
            if (P3e>1) e = e|P3ID;
            if (P4e>1) e = e|P4ID;

// Remove Bad Procs from working set
            e = GetState(WorkingSet)&(~e);
            PutState(WorkingSet,e);
// Reset Voter enable latches
            for (v=0; v<NumProcs; v++)
                if (e) SendValue(V[v],e);
//  Re-eval FTP
            FTPmodule_Eval();
           }//Recover
       }//CODE
}//FTP
```

Listing 9e. RML Model of Quad Processors plus Interfaces: FTP (contd)

```
 MODULE Interface {
    STATE IntState;
    RATE IntRate;
    RELATION Processor Proc: Pmess();
    RELATION InterfaceManager IntMan: ManMess();

// Not use of '&' instead of '=='.  This is because an Interface can be either
// GOOD or (GOOD|INUSE).  In either case it can fail.  The difference is the
// effect of failing if INUSE or not.  Also note that for this change the value
// of GOOD is made 1 and other State values modified accordingly.

    IF (GetState(IntState)&GOOD) TRANTO FailEffect() BY IntRate;

    CODE {

//FAILEFFECT
//  When an Interface fails it produces errors
       void FailEffect() {
          int s;
//  Have to retain INUSE bit
          s = GetNewState(IntState);
          PutState(IntState,((s&(~GOOD))|FAIL|ERROR));
          Interface_effect();
          }//FailEffect
```

Listing 9f. RML Model of Quad Processors plus Interfaces: Interface Module

```
//Pmess
//  When a processor FAILS the interface produces errors
       void Pmess(msg,who)
          int *msg,who; {
          Interface_FailEffect();
          }//Processor

//MANMESS
//  The Interface's Redundancy Manager can place the Interface Either
INUSE
//  or not INUSE
       void ManMess(msg,who)
          int *msg,who; {
          int s;
          s=GetState(IntState);
          switch (*msg) {
            case INUSE: PutState(IntState,s|INUSE); break;
            case REMOVED: PutState(IntState,s&(~INUSE)); break;
            }//switch
          }
       }
 }
```

Listing 9g. RML Model of Quad Processors plus Interfaces: Interface (contd)

```
   MODULE InterfaceManager {
     STATE IntManState;
     RATE IntManRec;
     RELATION Interface I[4]: Imess();

     IF (GetState(IntManState)==VULNERABLE) TRANTO
         RecEffect() BY FAST IntManRec;

     CODE {

/EVAL
// To evaluate Health of I/O system, only consider active (INUSE) elements
//  Notice that a simultaneous processor (e.g. P[0]) and interface (e.g. I[3])
//  failure causes the Manager to declare failure.  This is similar to
//  temporary exhaustion.
        void Eval() {
            int g,b,s,i;
            g=b=0;
            for (i=0; i<4; i++) {
               s = GetNewState(I[i].IntState);
               if (s&INUSE)
                 if (s&ERROR)
                   b++;
                 else
                   g++;
              }

            PutState(IntManState,GOOD);
            if (b>0) PutState(IntManState,VULNERABLE);
            if (g==0) PutState(IntManState,FAIL);
            }//Eval

//IMESS
//  When an Interface changes state, The parent manager must reevaluate
        void Imess(msg,who)
           int *msg,who; {
            InterfaceManager_Eval();
           }
```

Listing 9h. RML Model of Quad Processors plus Interfaces: InterfaceManager (contd)

```
//RECEFFECT
//  This recovery is pretty simple.
//  Interfaces which are INUSE and have ERROR are REMOVED.
//  If I[0] fails, I[1] is activated and if I[3] fails, I[2] is activated
        void RecEffect() {
           int s,i;
           for (i=0; i<NumProcs; i++) {
              s = GetState(I[i].IntState);
              if ((s&INUSE)&&(s&ERROR)) {
                SendValue(I[i],REMOVED);
                switch (i) {
                  case 0: if (GetNewState(I[1].IntState)&GOOD)
                                        SendValue(I[1],INUSE); break;
                  case 3: if (GetNewState(I[2].IntState)&GOOD)
                                        SendValue(I[2],INUSE); break;
                  }//switch
                }// if
              }//for

// Re-eval the Interface
           InterfaceManager_Eval();
           }//RecEffect
       }//CODE
 } //InterfaceManager
```

Listing 9i. RML Model of Quad Processors plus Interfaces: InterfaceManager (contd)

```
MODULE System {

     RELATION FTPmodule FTP;
     RELATION InterfaceManager IntMan;

     DEATHIF FTPFail();
     DEATHIF INTFail();

     CODE  {

//FTPFAIL
       int FTPFail() {
           return(GetState(FTP.FTPState)==FAIL);
           }//FTPFail

//INTFAIL
        int INTFail() {
           return(GetState(IntMan.IntManState)==FAIL);
           }//INTFAIL
      }//CODE
}//System

STATIC
{
#define GOOD 1
#define FAIL 2
#define REMOVED 4
#define VULNERABLE 8
#define ERROR 16
#define INUSE 32
#define P1ID 1
#define P2ID 2
#define P3ID 4
#define P4ID 8
#define NumProcs 4
}
```

Listing 9j. RML Model of Quad Processors plus Interfaces: System Module and Static Section

```
GLOBAL {
   Processor: P[4];
   Voter: V[4];
   Interface: Int[4];
   InterfaceManager: IntMan;
   FTPmodule: FTP;
   System: SYS;
   }

INIT {
   int i,j;
   for (i=0; i<NumProcs; i++) {
     Connect(FTP.V[i],V[i].FTP);
     Connect(P[i].Int, Int[i].Proc);
     Connect(V[i].PME,P[i].VME);
     for (j=0; j<NumProcs; j++)
         Connect(V[i].VSend[j],V[j].VReceive[i]);
     Connect(FTP.P[i],P[i].FTP);
     Connect(IntMan.I[i], Int[i].IntMan);
    }
    View(SYS.FTP,FTP);
    View(SYS.IntMan,IntMan);
   for (i=0; i<NumProcs; i++) {
     Set(P[i].ProcState,GOOD);
     Set(V[i].VoterState,GOOD);
     Set(V[i].VoterErrors,0);
     Set(V[i].VoterEnable,(P1ID|P2ID|P3ID|P4ID));
     }
   Set(Int[0].IntState,(GOOD|INUSE));
   Set(Int[1].IntState,GOOD);
   Set(Int[2].IntState,GOOD);
   Set(Int[3].IntState,(GOOD|INUSE));
   Set(IntMan.IntManState,GOOD);
   Set(FTP.FTPState,GOOD);
   Set(FTP.WorkingSet,(P1ID|P2ID|P3ID|P4ID));
   Rate(FTPmodule.FTPrec,3600.0);
   Rate(Processor.ProcFail,1.0E-4);
   Rate(Interface.IntRate,1.0E-5);
   Rate(InterfaceManager.IntManRec,3600.0);
   }
```

Listing 9k. RML Model of Quad Processors plus Interfaces: Global and Initialization Sections

Bibliography

[1] R. Butler. An abstract language for specifying markov reliability models. *IEEE Trans. on Reliability*, R-35(5):595–601, December 1986.

[2] G. Kernighan. Optimal sequential partitions of graphs. *Journal of the ACM*, 18(1):34–40, January 1971.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1992	Technical Memorandum

4. TITLE AND SUBTITLE
User's Guide to the Reliability Estimation System Testbed (REST)

5. FUNDING NUMBERS
WU 505-64-10-07

6. AUTHOR(S)
David M. Nicol , Daniel L. Palumbo, and Adam Rifkin

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23665-52255

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
NASA TM-107596

11. SUPPLEMENTARY NOTES
Nicol and Rifkin: College of William and Mary, Williamsburg, Virginia.
Palumbo: Langley Research Center, Hampton, Virginia.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Unclassified - Unlimited

Subject Category 38

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

The Reliability Estimation System Testbed is an X-window based reliability modeling tool that has been created to explore the use of the Reliability Modeling Language (RML). RML has been defined to support several reliability analysis techniques including modularization, graphical representation, Failure Mode Effects Simulation (FMES) and parallel processing. These techniques are most useful in modeling large systems. Using modularization, an analyst can create reliability models for individual system components. The modules can be tested separately and then combined to compute the total system reliability. Because a one-to-one relationship can be established between system components and the reliability modules, a graphical user interface may be used to describe the system model. RML has been designed to permit message passing between modules. This features enables reliability modeling based on a run time simulation of the system wide effects of a component's failure modes. The use of failure modes effects simulation enhances the analyst's ability to correctly express system behavior when using the modularization approach to reliability modeling. To alleviate the computation bottleneck often found in large reliability models, REST has been designed to take advantage of parallel processing on hypercube processors.

14. SUBJECT TERMS
Reliability analysis, Failure modes effects, Analysis, Parallel processing

15. NUMBER OF PAGES
90

16. PRICE CODE
A05

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		