Model Checking RSML^{-e} Requirements*

Yunja Choi and Mats P.E. Heimdahl
Department of Computer Science and Engineering, University of Minnesota
200 Union Street S.E., 4-192, Minneapolis, MN 55455, USA
E-mail: {yuchoi,heimdahl}@cs.umn.edu

Abstract

Model checking is a promising technique for automated verification or refutation of software systems. Nevertheless, it has not been used widely in practice mainly due to the lack of the supporting tools that incorporate the model checking activity into the development process. As a part of our overall method supporting specification centered system development, *we have implemented a translator between a formal specification language RSML^{-e} and a symbolic model checker NuSMV.*

Our translation and abstraction approach aims at usability in practice so that model checking can be used as a routine process during requirement analysis without requiring much knowledge about formal methods. Preliminary result from applying the system in a commercial setting is quite promising. In this paper, we discuss our translation and abstraction approach in some depth and illustrate its feasibility with some preliminary results.

1 Introduction

Automated verification of software systems using model checking techniques has been an active research area for quite some time [1, 4, 5, 14, 16]. Although model checking programs is feasible for small size programs with some limitations on the program constructs used [4, 14, 16], model checking requirement specifications has seen better success in practice; specifications are more abstract than programs and the specification languages do not typically contain language features complicating model checking, for example, threading, recursion, and dynamic memory management [2, 9, 16]

To support a specification centered software engineering process, we have defined and built a translator from a fully formal specification language RSML^{-e} [17, 25] to a symbolic model checker NuSMV [23]. The translator provides practitioners with a push-button verification process once the requirements are specified using RSML^{-e}. In this paper we report on our translation approach and discuss the challenges we faced with some unique aspects of RSML^{-e}. We also discuss some preliminary experiences with using this approach on an industrial system.

Our goal is *practical* application of formal methods—we want practitioners to be able to read and understand the formal models, be able to create models with methodological guidance [26], be able to execute the models and use them as early prototypes [25], and perform formal analysis, all without necessarily requiring detailed knowledge of formal methods. This quest for simplicity is reflected in our approach to model checking RSML^{-e} specifications.

When we started the model checking project, we set the following goals: (1) the translation must to the largest extent preserve the full power of the source language RSML^{-e}, (2) the translation must require little user interaction, (3) the translation must closely match the structure of RSML^{-e} so that counterexamples are easy to interpret without detailed knowledge of the translation process, and, finally, (4) any abstractions must be fully automated. Although much work remains to be done—especially with respect to abstraction—preliminary results indicate that we are on a promising track to achieving our goals.

We will first discuss our general verification framework and the languages involved, RSML^{-e} and the specification langauge for NuSMV. We describe our general translation approach in Section 3 and discuss some abstractions necessary to map an RSML^{-e} specification to NuSMV. Some preliminary results from our collaboration with industry appears in Section 5 followed by brief discussion and conclusion.

2 Framework

Figure 1 shows an overview of our verification framework. The user models the required behavior of a system in the fully formal and executable specification language RSML^{-e} . After evaluating the functionality and behavioral

*This work has been partially supported by NASA grant NAG-1-224 and NASA contract NCC-01-001.

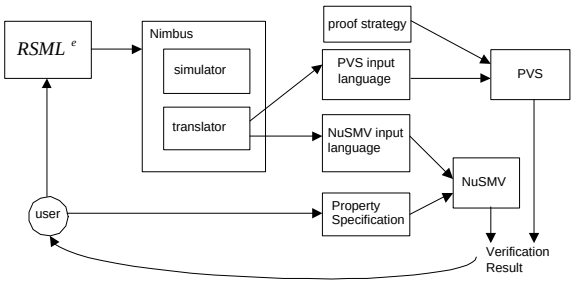


Figure 1. Verification Framework.

correctness of the specification using the NIMBUS simulator, users can translate the specifications to the PVS or NuSMV input languages. For model checking the system model, users specify required properties in a temporal logic, CTL or LTL, and use NuSMV for the analysis.

To make it easier for the reader to understand our translation approach, we provide a short overview of our source language RSML^{-e} as well as the target language of NuSMV.

2.1 RSML^{-e}

RSML^{-e} is based on the Statecharts like language Requirements State Machine Language (RSML) [19]. RSML^{-e} is different from RSML (and Statecharts) primarily in that RSML^{-e} supports rigorous specifications of the interfaces between the environment and the control software, and RSML^{-e} is a synchronous data-flow language without any internal broadcast events.

An RSML^{-e} specification consists of a collection of input variables, state variables, input/output interfaces, functions, macros, and constants; *input variables* are used to record the values observed in the environment, *state variables* are organized in a hierarchical fashion and are used to model various states of the control model, *interfaces* act as communication gateways to the external environment, and *functions and macros* encapsulate computations providing increased readability and ease of use.

Figure 2 shows a specification fragment of an RSML^{-e} specification of a simple Altitude Switch System; the figure shows the definition of one state variable—the *AltitudeStatus*. Figure 3 shows how the hierarchical variables would be visualized in the NIMBUS tools[1].

The state variable *AltitudeStatus* is declared as a child state of *PowerStatus* and is active only when the variable *PowerStatus* has the value *On*—this notion of hierarchical variables provides the same abstractions and structuring mechanism as the AND and OR states in Statecharts [15], but the semantics is much simpler.

[1] A pretty printer producing cross referenced and indexed LaTeX documents is available in NIMBUS. Nevertheless, for space reasons we will use the more compact ASCII representation in this paper.

```
STATE_VARIABLE AltitudeStatus :
  VALUES : { Unknown, Above, Below, AltitudeBad }
  PARENT : PowerStatus.On
  INITIAL_VALUE : UNDEFINED
  CLASSIFICATION : State

  EQUALS Unknown IF
    TABLE
       ivRese                                          : T *;
       PREV_STEP(..AltitudeStatus) = UNDEFINED  : * T;
    END TABLE

  EQUALS Below IF
    TABLE
       BelowThreshold()     : T ;
       AltitudeQualityOK()  : T ;
       ivReset              : F ;
    END TABLE

  EQUALS Above IF
    TABLE
       AboveThresholdHyst()  : T ;
       AltitudeQualityOK()   : T ;
       ivReset               : F ;
    END TABLE

  EQUALS AltitudeBad IF
    TABLE
       AltitudeQualityOK()  : F ;
       ivReset              : F ;
    END TABLE

END STATE_VARIABLE
```

Figure 2. A piece of RSML^{-e} specification.

The conditions under which the state variable changes value are defined in the EQUALS clauses in the definition. The tables are adopted from the original RSML notation– each column of truth values represents a conjunction of the propositions in the leftmost column (a '*' represents a "don't care" condition). If a table contains several columns, we take the disjunction of the columns; thus, the table is a way of expressing conditions in a disjunctive normal form. *PREV_STEP(..AltitudeStatus)* refers to the previous (old) value of *AltitudeStatus* while *ivReset* without the *PREV_STEP* operator refers to the current (new) value of the variable.

In addition to state variables, RSML^{-e} allows an analyst to specify how the system interacts with its environment; the way a system receives the value of input variables and signals outputs to its environment can be specified using input and output interface constructs in RSML^{-e} . Figure 4 shows an example of interface specification for the Altitude Switch System; the input receiver *AltitudeInterface* receives and assigns values of input variables whenever the *receive_handler* condition (the condition TRUE means that a message arrives at the interface) satisfies. Otherwise, the default handler (the second handler with no name) condition determines the value of input variables. Similarly, output sender *FaultDetectionInterface* sends out the $fault$ message whenever the handler condition satisfies (at every *200 ms* in this case).

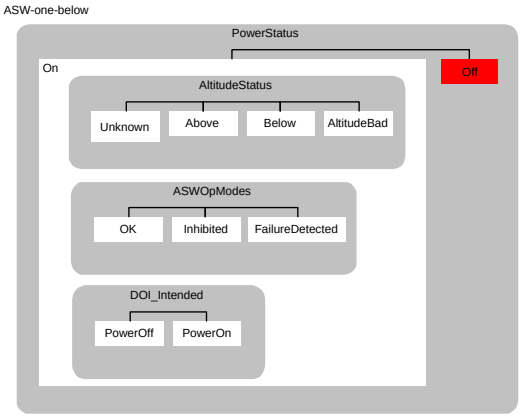


Figure 3. A graphical view of RSML^{-e} model.

A few key characteristics of RSML^{-e} that sets it apart from related notations can be summarized as follows:

```
TYPE_DEF AltitudeQualityType { Good, Bad }

MESSAGE AltInMessage {
  a1 IS INTEGER,
  alt1q IS AltitudeQualityType}

IN_INTERFACE AltitudeInInterface :
  MIN_SEP : UNDEFINED
  MAX_SEP : UNDEFINED
  INPUT_ACTION : RECEIVE(AltInMessage)

  RECEIVE_HANDLER :
    CONDITION : TRUE

    ASSIGNMENT
      Altitude := a1,
      Altitude_Quality := alt_q
    END ASSIGNMENT
  END HANDLER

  HANDLER
    CONDITION:
      TABLE
          TIME:AltitudeInterface::LAST_IO > 200 MS : T;
      END TABLE
    ASSIGNMENT
      Altitude:=UNDEFINED
      Altitude_Quality:=UNDEFINED
    END ASSIGNMENT

END IN_INTERFACE
```

```
MESSAGE FaultMessage {
  fault IS BOOLEAN
}

OUT_INTERFACE FaultDetectionInterface :
  MIN_SEP : 50 MS
  MAX_SEP : 200 MS

  OUTPUT_ACTION : SEND(FaultMessage)

  HANDLER :
    CONDITION :
      TABLE
          TIME-FaultDetectionInterface::LAST_IO>200MS : T;
      END TABLE

    ASSIGNMENT
      fault := FaultDetectedVariable
    END ASSIGNMENT

    ACTION : SEND
  END HANDLER
END OUT_INTERFACE

IN_VARIABLE Altitude : INTEGER
  INITIAL_VALUE : Undefined
  UNITS : ft
  EXPECTED_MIN : 0
  EXPECTED_MAX : 40000
END IN_VARIABLE

IN_VARIABLE Altitude_Quality : AltitudeQualityType
  INITIAL_VALUE : Undefined
END IN_VARIABLE
```

Figure 4. RSML^{-e} interface specifications

Support for interface specifications: RSML^{-e} allows a rigorous specification of how the model of the controller interacts with the environment. This allows an analyst to refine a model from a very high-level model relying on "ideal" sensors and actuators to a more detailed model taking the idiosyncracies of the hardware into account.

Support of macros and functions: The use of macros and functions increases writability, readability, and reusability of specifications. For example, a flight guidance system may have a very complex mode logic that requires dozens of lengthy condition tables. By encoding each condition table with a macro, one can not only improve the readability of the specifications but also help localize errors and future changes.

Data-flow semantics: RSML^{-e} transitions are purely condition-based and free of internal events—as soon as the guards in a variable definition can be evaluated, it will take on its new value. The variables are partially ordered based on the data dependency induced by the guard conditions—a similar semantics is adopted in the programming language Lustre [13]. Data-flow semantics removes complex issues caused by internal events, such as infinite triggering events or analysis of micro-steps [5], from the language.

Use of Undefined *values*: Startup behavior and behavior in the face of sensor failures pose particular challenges when specifying control systems—under these circumstances we simply do not know what the state of the environment might be. RSML^{-e} supports modelling of this uncertainty by providing the concept of *Undefinedness*. One can explicitly specify the initial value of variables at startup to as *Undefined*, such as Altitude=UNDEFINED in Figure 4. Also, when a parent variable takes on a new value, each child variable of the parent value that was just changed are no longer relevant and must not be used—these child variables are *Undefined*. RSML^{-e} supports for both explicit and implicit *Undefinedness*.

2.2 NuSMV

NuSMV [23] is a symbolic model checker evolved from the Carnegie Mellon University (CMU) version of SMV [22]. NuSMV provides for modelling hierarchical descriptions, and for the definition of reusable components. NuSMV supports verification of properties expressed in both temporal logics CTL(Computational Tree Logic) and LTL(Linear Time Logic).

The input language of NuSMV is designed to allow the description of synchronous or asynchronous finite state machines at various levels of abstraction. The language of NuSMV is structured using keywords MODULE, VAR, IVAR, DEFINE, ASSIGN, TRANS, INVAR, and SPEC; MODULE represents a unit of a reusable component, VAR and IVAR are for variable declaration, DEFINE is for macro-like symbolic representation of expressions, ASSIGN and TRANS[2] are to specify transition relations, INVAR is to specify system invariants, and SPEC is to specify system properties in temporal logic.

Figure 5 shows a small NuSMV model. The state variables and input variables, which determine the state space, are declared under the VAR keyword. Since NuSMV is limited to describing finite state machines, only finite data types

[2] Since TRANS statements are more expressive but less robust than ASSIGN statements [5], we only consider ASSIGN statements in our work.

```
MODULE main
VAR
  request : boolean;
  state : {ready, busy};
  counter : 0..10;
  m_check_busy : busy_status(request, state);

ASSIGN
  init(state):=ready;
  next(state):=
      case
          (state=ready & request=1)  | (m_check_busy.status=1)  : busy;
            1                                                   :ready;
      esac;
  ...

SPEC
  AG(request -> AF state=busy)

MODULE busy_status(request, state)
VAR
  status : boolean;

ASSGIN
  init(status):= 0;
  next(status):= (state=busy & next(request)=1);
```

Figure 5. A sample NuSMV program

are provided, such as Boolean, enumerations, integer subrange, and fixed length array.

The initial value (represented by $init$ expression) and the transition relation of a variable (using $next$ expression) can be specified by a collection of simultaneous assignments using the ASSIGN keyword; all initial-state assignments are executed simultaneously at system start-up and all next-state assignments are executed simultaneously each system cycle. As shown in Figure 5, the next-state transition assignment is often defined using a $case$ expression. NuSMV evaluates and picks the first condition satisfied. Case '1'(representing $true$) is the default case.

There must be one MODULE named $main$ for each system model; modules with other names represent synchronous/asynchronous sub-processes or reusable components of the system.

Temporal logic formulas that specify required properties of the system model are captured using the keyword SPEC. In this example, the formula specifies that every time *request* is true, then in all possible futures, eventually *state* must become *busy*.

The next few sections will discuss how we translate RSML^{-e} to NuSMV to take advantage of the powerful model checking support provided in the NuSMV tools.

3 Translation from RSML^{-e} to NuSMV

This section is split into two. In the first part, we introduce the basic translation to give the reader a flavor of our approach. This translation scheme is quite generic for state machines of this type and is similar to the one selected in [5] except for few translation choices. The challenges faced when translating various RSML^{-e} specific constructs are discussed in the second half.

3.1 Basic Translation

The basic construct in RSML^{-e} is the variable and the next-state relation for the variable. If we disregard some issues with the types in RSML^{-e} (for example, time types and the notion of Undefined) and the complexity added by interfaces with the environment, the basic translation to NuSMV is quite straight forward.

Types and variables: RSML^{-e} supports basic data types, such as boolean, enumerated, integer and real, and user-defined types based on the basic types. Since NuSMV does not support user defined types, all reference types are converted to one of boolean, enumeration, or integer subrange types. Input and state variables are translated using the NuSMV keyword "VAR" as "VAR identifier_name : var_type", where the var_type can be boolean, enumeration, or integer subrange. For example, the state variable *AltitudeStatus* from Figure 2 and the input variable *AltitudeQuality* from Figure 4 will be translated to

```
VAR
 AltitudeQuality : {Good, Bad};
 AltitudeStatus : {Unknown, Above, Below, AltitudeBad};
```

Transitions: RSML^{-e} transition conditions are translated to assignment expressions in NuSMV. For example, the transition relation defined using the EQUALS expression in the definition of state variable *AltitudeStatus* in Figure 2 is translated to

```
ASSIGN
 next(AltitudeStatus):=
  case
      AltitudeStatus = Un_Defined |
            next(ivReset)                  : Unknown;
      next(m_BelowThreshold.result) &
            next(m_AltitudeQualityOk.result) &
            ! next(ivReset)                : Below;
      next(m_AboveThresholdHyst.result) &
            next(m_AltitudeQualityOK.result) &
            !next(ivReset)                 : Above;
      !next(m_AltitudeQualityOk.result) &
            !next(ivReset)                 : AltitudeBad;
      1                                    : AltitudeStatus;
  esac;
```

where m_XX.result, such as m_BelowThreshold.result, is a reference variable of the sub-module defining a macro. The translation of macros and dealing with *Undefinedness* will be described shortly.

In the RSML^{-e} semantics, an expression with a PREV prefix indicates the *old* value of a variable–the values the variable had before we started the evaluation of the state transition relation and an expression without a PREV prefix indicates the *new* value of a variable as computed in the current step. We view these expressions with and without PREV

prefix correspond to the NuSMV expressions without and with the *next* prefix, respectively, as shown in Figure 6 (a). In this interpretation, we compute a new value of a variable x, i.e., $next(x)$, based on the old values of variables and new values of other variable which x depends on.

Another translation approach is possible [5]; in Figure 6 (b), the next-state value is computed based on the current-state and the previous-state values. In this case, a separate variable $prev\_x$ is declared with the same type as x and the transition relation is maintained by assigning $next(prev\_x) := x$ at each time $next(x)$ is evaluated. (pointed by the dashed arrow in the figure).

For example, if the value of a variable x is computed based on the previous value of x and the value of y, i.e. $x = f(prev(x), y)$, our approach translates the transition relation to $next(x) := f(x, next(y))$, whereas the alternative approach translates it to $next(x) := f(prev\_x, y)$ introducing an additional variable $prev\_x$. The choice of translation approach for PREV expressions is closely related to the choice of translation for macros, which will be discussed shortly.

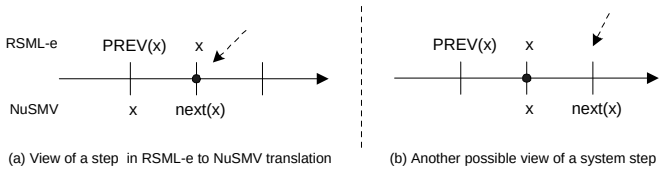
Macros and Functions: Our translation approach translates each macro/function to a NuSMV sub-module and declares one reference variable for the sub-module in the main module. For a simple example, a macro *DOI_Tobe_On* defined as follows

```
MACRO DOI_Tobe_On() :
    TABLE
        @T(..DOI_Intended = PowerOn) : T;
        inhibitSignal=inhibit        : F;
    END TABLE
END MACRO
```

will be translated to

```
MODULE DOI_Tobe_On(DOI_Intended)
 VAR
  result : boolean;
 ASSIGN
  init(result):= 0;
  next(result):= ! (DOI_Intended = PowerOn) &
                  (next(DOI_Intended)=PowerOn ) &
                 ! (next(inhibitSignal)=inhibit);
```

and a reference variable will be declared in the main module (or the module which refers to the macro) as follows:



(a) View of a step in RSML-e to NuSMV translation (b) Another possible view of a system step

Figure 6. Different views in system steps

```
MODULE main
 ...
 VAR
  DOI_Intended : {PowerOn, PowerOff };
  m_DOI_Tobe_On : DOI_Tobe_On(DOI_Intended);
 ...
```

Here, @T(a) is a SCR-style expression that is equivalent to {not PREV_STEP(a) & a}. The boolean variable *result* is declared in the sub-module to represent the value of the macro. Note that NuSMV does not allow global variables, and thus, every variable referred from a sub-module is passed as a parameter.

Function definitions in RSML^{-e} is translated in a similar way except that the type of the variable *result* is the same as the function type and the initial value of the function is unconstrained. With this very simple translation, the bulk of an RSML^{-e} specification of a proposed system can be captured in NuSMV.

The macro/function translation introduces one additional variable $result$ per macro/function which is not desirable for model checking. An alternative way to save the number of variables after translation is to use the DEFINE declaration when the macro does not have parameters. DEFINE declaration does not introduce any additional variables but it can be used only for simple expressions–conjunctions and/or disjunctions of variables without $next$ expressions. In other words, DEFINE declaration cannot be used for an expression containing *next* expressions, and thus, is not usable in our case since our translation maps all *non-prev* expressions to NuSMV *next* expressions to reduce the state space.

In our preliminary application, a specification for a flight guidance system contains 115 macros and *prev* expressions for almost every state and input variables (total 84). Considering that state or input variables can be of enumerate or numeric types, 84 additional *prev* variables can be more costly than 151 additional boolean variables. The trade-offs need to be investigated further for better management of the state space.

3.2 Mapping RSML^{-e} specific constructs

RSML^{-e} has some specific constructs, such as interfaces and *Undefined* values that require special attention.

Translation of Interfaces and Messages: An RSML^{-e} model of an embedded system interacts with its environment via interfaces by receiving (reading) messages on input interfaces or by sending (publishing) messages through output interfaces. The actual input values are assigned only when messages arrive in the input interfaces and the interface handler conditions are satisfied—the handlers allow the specifier to handle messages differently depending on the

content of the message (for example, if values are out of bounds) or the timing of the message (for example, if it arrived early or late). Output interfaces assign and send out messages only when output handler conditions are satisfied.

In Figure 4, an output interface *FaultDetectionInterface* specifies that the interface shall assign and send out the message *fault* at every *200 ms*. This type of specification allows the user to capture the details of the system/environment interactions. This also allows us to check desired properties related to the exchange of information between system and environment. For example, a required property can be "the system sends out the fault message whenever the fault is detected in the system".

RSML^{-e} contains several expressions that are related to the interfaces, for example, to check if a message has arrived at an input interface, to check if an output has been sent, and to determine the time since a message was sent. In translation of interface related expressions, boolean variables are declared to represent the action of the interface and the satisfaction of each handler condition.

For the *FaultDetectionInterface* example, a boolean variable *FaultDetection_OS* is declared to represent the sending action, and a boolean variable *FaultDetection_OSH* is declared to represent the handler condition as follows.

```
VAR
 fault : boolean; /* for message field */
 FaultDetection_OS : boolean; /* output sending flag */
 FaultDetection_OSH : boolean; /* for handler flag*/
```

The interface does not send out messages in the initial state, thus, the initial value of the output sending flag is set to false. Initial values for the message and the handler condition flags are unconstrained since the initial values do not affect the sending action once we set the sending flag initially.

Interface action and handler conditions constrain each other; a handler condition for an input interface is evaluated based on the interface message arrival event (or absence of the message arriving), the action of an output interface is determined by the evaluation of its handler conditions. As shown below, the next value of the *fault* variable is evaluated whenever the handler condition is satisfied and remains the same otherwise. The send action becomes true whenever the handler condition becomes true and becomes false otherwise.

```
ASSIGN
 init(FaultDetection_OS):=0;
 next(fault):=
      case
        next(FaultDetection_OSH) : next(FaultDetectedVariable);
        1                        : fault;
      esac;
 next(FaultDetection_OS):=
      case
        next(FaultDetection_OSH): 1;
        1                       : 0;
      esac;
```

With this translation, the property "the system sends out the fault message whenever the fault is detected in the system" can be specified in CTL as *AG(FaultDetectedVariable → FaultDetection_OS)*, which turns out false because of the week handler condition; the fault can be detected within the $200\ ms$ time window but the interface does not send out the message until the time threshold reaches.

Translation for the handler condition *FaultDetection_OSH* will be discussed in the next section along with the abstraction for time expressions. Translation for input interfaces is similar to output interface and is omitted here to save space.

Flattening the hierarchy and Undefinedness: As in RSML and Statecharts, RSML^{-e} supports the creation of hierarchical models. While RSML supports state hierarchies by defining super-state and sub-state relations, the hierarchy in RSML^{-e} is specified by defining the value of the parent variable for each child variable—RSML^{-e} makes no distinction between variables and states. In Figure 2, the hierarchical relationship between the state variables *PowerStatus* and *AltitudeStatus* is represented by specifying the PARENT field of the *AltitudeStatus* definition—*AltitudeStatus* is relevant only if *PowerStatus* has the value *On*.

In translation, the hierarchy is flattened and each state variable is declared as a separate variable regardless of whether it is a parent variable or a child variable. Each child state variable declaration, however, includes a specific value *Un_Defined* to designate that it may not be relevant since its parent variable does not have the right value.

```
VAR
 PowerStatus : {On, Off};
 AltitudeStatus :
     {Unknown, Above, Below, AltitudeBad, Un_Defined};
```

The transition relation for the child variable needs to reflect this passive *Undefinedness*. For each transition condition, whether its variable has the right value is checked first; if the parent variable has the wrong value, its child variable(s) are always set to *Un_Defined*. If the parent has the right value, the child variable is relevant and its transition relation will be evaluated normally. When taking *Undefinedness* into account, the transition relation for the *AltitudeStatus* state is translated to

```
ASSIGN
 next(AltitudeStatus):=
  case
      ! (next(PowerStatus)=On)                 : Un_Defined;
      AltitudeStatus = Un_Defined |
           next(ivReset)                       : Unknown;
      next(m_BelowThreshold.result) &
           next(m_AltitudeQualityOk.result) &
```

```
        ! next(ivReset)                 : Below;
    next(m_AboveThresholdHyst.result) &
        next(m_AltitudeQualityOK.result) &
        !next(ivReset)                  : Above;
    !next(m_AltitudeQualityOk.result) &
        !next(ivReset)                  : AltitudeBad;
    1                                   : AltitudeStatus;
esac;
```

The *Undefined* value in RSML^{-e} has a dual purpose. First, as described above, *Undefined* is used to represent state variables that are not relevant because the part of the hierarchy where they reside is not active. Second, *Undefined* is used to specify cases when the exact value of a system variable simply is unknown, for example, at system startup.

Data type extension is required in translation since the *Undefined* is an intrinsic data value in the RSML^{-e} semantics. For enumerated variables that require the *Undefined* value, simple type extension is used to explicitly include the value *Un_defined*. On the other hand, the *Undefined* value of a numeric variable is captured by declaring a separate variable to designate the *Undefinedness* of the variable since the integer type in NuSMV cannot simply be extended with the *Undefined* value.

4 Abstraction

In the previous section, we presented a simple and direct translations from RSML^{-e} to NuSMV without introducing much abstraction. Our focus was on efficient automation and the usability aspect of the translated model; direct translation largely preserves the structure of original specifications so that the user can understand the translated model with minimum effort, and more importantly, can understand the counter examples generated by NuSMV. Nevertheless, direct translation is not always possible for certain aspects of RSML^{-e}, for example, global time variables and numeric variables must be abstracted away since NuSMV allows only variables over finite domains.

The purpose of abstraction is twofold; (1) to define a semantic preserving translation for those constructs in RSML^{-e} that cannot be directly translated to NuSMV and (2) to reduce the number of state variables to avoid the state space explosion problem. Together with previously published abstraction techniques suggested for software model checking [1, 5, 7, 18], our translation adopts three major abstractions, two of which are rather specific to RSML^{-e} because of its inclusion of interfaces and time. In the following paragraphs we will describe time abstraction, interfaces and message abstraction, and abstraction of numeric variables.

Time abstraction: RSML^{-e} views time as an external clock globally visible within a model. Since TIME is an infinitely increasing variable with undefined precision, it must be abstracted away in some way. All RSML^{-e} time expressions are in terms of absolute time. The time of all events of the system's history are available through the various RSML^{-e} TIME expressions, such as the time of a message arrival or the time of change of the value of a state variable. For example, the expression TIME refers the current time value and *FaultDetectionInterface::LAST_IO* represents the last time an exchange happened on the interface.

Since NuSMV operates on finite state machine and is not able to handle infinite time, the absolute time is abstracted to relative time. In the example of the previous section, the guard condition in the handler will be true after the duration since the last IO exceeds the threshold of *200 ms*. Only the fact that the duration exceeds the threshold is of importance; after time has exceeded the threshold, the length of the duration is no longer important.

To map the absolute time in RSML^{-e} to the relative time that can be represented in a model checker, we declare a separate timer for each time expression. In our example, the time variable *t_FaultDetection* is declared as an integer variable between 0 to 201 to represent the *TIME - FaultDetectionInterface::LAST_IO* $>$ *200 MS* expression. The value of *t_FaultDetection* is set to zero initially, increases by one at every system cycle, and remains the same when it exceeds the threshold. It gets reset to zero after a message is sent out.

```
VAR
 t_FaultDetection: 0..201;
ASSIGN
 init(t_FaultDetection):= 0;
 next(t_FaultDetection):=
     case
         /* reset the timer after the message sent out */
         FaultDetection_OS      : 0;
         /* remain same after exceeding the threshold */
         t_FaultDetection > 200 : t_FaultDetection;
         /* increases by one by default */
         1                      : t_FaultDetection + 1;
     esac;
```

The handler condition flag *t_FaultDetection_OSH* becomes true when *t_FaultDetection* is greater than 200; *next(t_FaultDetection_OSH):=(next(t_FaultDetection)* $>$ *200)* .

Note that we do not need to use the time granularity of *1 ms* for the relative timer; depending on the time resolution necessary for verification, we can use *2*, or *20* for the threshold by dividing it by factor of 100 or 10. Nevertheless, abstraction for several timing expressions has to be done carefully to preserve the relative time granularity. For example, the two time expressions *TIME -InterfaceA::LAST_IO* $>$ *200 ms* and *TIME - InterfaceB::LAST_IO* $>$ *5 ms* can be abstracted as *t_InterfaceA* $>$ *200* and *t_InterfaceB* $>$ *5* but uniformly dividing the threshold values by the factor of *10* or *100* does not work in this case. Instead, the abstracted timing threshold can be obtained by dividing each threshold value by the

greatest common divisor of all threshold values in the specification, i.e., *gcd(200, 5) = 5* abstracting the time expressions to *t_InterfaceA* > 40 and *t_InterfaceB* > 1.

Abstracting interfaces and messages: Since model checking suffers from the state space explosion problem, reducing the number of state variables as much as possible is crucial for the success of automated translation and verification.

Support for specification of interfaces and messages is one major characteristic of RSML^{-e} that makes rigorous specifications of system level inter-component communication possible. Nevertheless, many safety properties or essential system requirements can be verified regardless of the interface definitions. Therefore, we can optionally remove the interface and message field portion from the translation process reducing the number of variables necessary to capture a model in NuSMV. In addition, selective translation of interfaces and messages can be done by *dependency checking* on expressions; an interface and its corresponding messages are translated only when an expression in a transition condition refers to interface related variables, such as interface timers.

The selective interface translation is different from the *cone of influence reduction* [8] built into most symbolic model checking tools; the cone of influence reduction removes all variables that are not relevant to the property in question while the selective translation removes redundant interfaces that have no effect on the change of state variables. For example, suppose an input variable x gets assigned through an input interface I via a message m and the change of state variable S is dependent on the value of x. When a property in question is related to the value of S, the cone of influence reduction will keep all the variables that have a data dependency relation with S including $\{I, m, x\}$. The selective translation keeps only the input variable x if the transition of S does not depend on any interface related expressions.

Numeric variable abstraction: Besides time variables, translation of numeric variables such as integer or real variables requires various types of abstraction. Various abstractions have been suggested to deal with finite or infinite numeric variables [2, 3, 10, 11, 21] mostly with the aid of decision support tools such as PVS [24]. Currently, we plan to handle integer variables using *domain reduction abstraction* [6], which can be performed during translation in conjunction with other existing abstraction techniques. Our goal is to remove user guidance from the abstraction process as well as to remove the large variable domains without affecting observable system behavior.

The *domain reduction abstraction* technique reduces a large variable domain to a small finite one. The basic idea is to selectively choose representative values from each data equivalence class as partitioned by the guarding conditions over numeric variables. For example, when a system has a data variable x over the domain $[-1000, 1000]$ and guarding conditions $x < 10$ and $x > 250$, a reduced domain $\{9, 11, 251\}$ is sufficient to simulate the original system assuming that the system is not data sensitive. A detailed discussion of this abstraction technique is beyond the scope of this report—detailed theoretical support and examples can be found in [6].

system model	number of variables	verification			counter example generation		
		system time	user time	memory	system time	user time	memory
FGS model version 0.3	150	0.44 s	76.74 s	39 M	2.3 s	2853.11 s	77 M
FGS model 0.3 (a)	150	0.03 s	13.21 s	10.8 M	0.04 s	25.7 s	10.8 M
FGS model 0.3 (b)	87	0.04 s	5.35 s	10 M	0.05 s	8.21 s	10.2 M
FGS model version 0.5	210 (326)	0.62 s	339.47 s	55 M	7.51 s	9719.0 s	290 M

Figure 7. Sample system usage data.

5 Preliminary Results

We have implemented the translation scheme discussed in this paper in the the NIMBUS tools and are currently evaluating the scalability of the verification approach as well as its ease of use by practitioners. Preliminary results are quite promising.

In collaboration with Rockwell-Collins Advanced Technology Center we have modelled and analyzed a collection of Flight Guidance Systems (FGS). The FGS compares measured position, speed, and altitude to the desired state of the system and generates pitch and roll guidance commands to minimize the difference between the measured and desired state.

Rockwell-Collins has developed a collection of progressively more complex RSML^{-e} FGS models, ranging from a simple model (FGS 0.0 – 600 lines of RSML^{-e} with 21 macros, and 45 variables after translation) to a detailed FGS model including numeric variables and timing constraints (FGS 0.5 – 3832 lines of RSML^{-e} with 119 macros and functions, and 326 variables after translation)[3]. The proprietary nature of the models prevent us from discussing details at this early stage in the project, but we can provide some preliminary results regarding the scalability of the approach. All results are obtained on a Linux workstation with 800 MHz CPU and 512 M of memory with 99.9% CPU usage during the verification.

Figure 7 illustrates a sample system usage for verification and counter example generation for FGS models. The

[3]These models have been evolved since this paper was first written. The experiment data is based on the models and the translator as of April 2002

first row shows the result for the FGS model 0.3 before applying abstraction on numeric variables. The second row (row a) shows the result after abstracting numeric variables using domain reduction abstraction and the third row (row b) is the result after abstracting numeric variables and interfaces. The last row shows the usage data for verification and counter example generation on FGS model 0.5, the largest model we have so far, after application of abstractions. The translated model of the FGS 0.5 without abstraction includes 326 variables and the system runs out of memory when checking properties over the model. Translation with abstractions produces a model with 210 variables which is feasible for model checking.

In the figure, the second column represents the total number of variables in the translated model regardless of variable types. System time represents the time used for system interrupt and user time represents the total execution time for the verification and counter example generation. The cone of influence reduction built into NuSMV is used in all cases. The performance data for the columns marked "verification" are results from a property that is verified to be true. The data in the columns labelled "counter example generation" are from a property that turned out false.

Note the number of variables remains same before and after the abstraction for numeric variables for the FGS version 0.3 even though the usage data shows a large difference in performance. The difference is unsurprisingly mainly due to the abstraction of integer variables with large domains. The FGS model version 0.3 has 8 integer variables (3 of which have infinite domains). Our direct translation maps a numeric variable with infinite domain to an integer variable with a finite, but very large, domain, so that these models can be model checked. In this case, however, the FGS model is data insensitive, i.e., the behavior of the model is not affected by specific data values. Therefore, domain abstraction reduces the large domain of the integer variables to a single value.

Model checking the Flight Guidance Systems written in RSML^{-e} has scaled surprisingly well this far. We expected that the high fidelity of our translation would lead to NuSMV models that simply could not be model checked without aggressive abstraction. This has not been the case. Currently, model checking the full FGS specifications is feasible by applying domain reduction abstraction and removal of unneeded interfaces, but the size of the specification is gradually growing as more details are added and we will need to incorporate more aggressive abstractions to complete the project.

In addition, by using assume-guarantee-style modular verification techniques [12, 20], we can verify portions of a specification and then combine the results to a statement about the specification as a whole. Our research group is working on adding a module construct to RSML^{-e} to support specification reuse. This construct will also enable us to support modular verification. The goal of our efforts is still to include only abstractions that can be invoked with minimal user intervention and with limited knowledge of the theory behind model checking—we want the analysis techniques to be usable by practitioners and full automation is, in our opinion, an absolute must.

6 Discussion

Automatic translation of formal specifications for model checking purposes is not a new idea. Chan et al. [5] focused on the feasibility of model checking software requirement specifications and suggested translation rules from RSML (the predecessor language to RSML^{-e}) to SMV. Heitemeyer et al. [18] discuss abstraction as part of the translation and verification of SCR specifications. Nevertheless, application of model checking to industry sized applications is still rare.

Compared to those previous works, our translation approach makes several different choices, such as our treatment of *prev* expressions and macros, but the simplicity of our translation is mainly due to the removal of internal events from the specification language.

Transitions in RSML or Statechart are based on events in the form of *trig[cond]/acts* where *trig* is a trigger event, the guarding condition *cond* is a predicate on states and inputs, and *acts* is a set of action events. One trigger events can cause cascading action/trigger events, and thus, an input event in one step can cause several (or possibly infinitely many) micro-steps of actions according to the cascading action/trigger events.

Automatic translation of specifications with internal events is more difficult than for data flow models mainly due to the potentially complex event propagation. RSML^{-e} avoids this type of problems by removing the internal events from the language; yet, the expressive power of RSML^{-e} has been sufficient in all projects where it has been used.

7 Conclusion

We have implemented a translator to support the practical application of automated verification of software specifications. The initial version of the translator maps the source language (RSML^{-e}) quite closely onto the target langauge (NuSMV) with only minor abstractions. In the first step of the project we wanted to know how well this approach would scale without the introduction of aggressive abstractions. With few abstractions, the analysis results are very accurate—spurious counter examples are rare—making the analysis appealing to practitioners.

We have largely satisfied the goals we set at the beginning of the project; the translation is fully automated and

preserves (almost all of) the expressive power of RSML^{-e}, the counterexamples are easy to interpret since the structure of the NuSMV model is very close to the RSML^{-e} model, and the abstractions we have implemented are intuitive and easy to invoke.

Our next step is to anticipate future problems as the size of the specification grows. We are investigating alternative translation approaches, such as different treatment of macros, and more aggressive abstraction techniques.

Acknowledgements: We would like to thank Steve Miller, David Lempia, and Alan Tribble at Rockwell-Collins' Advanced Technology Center for building the FGS models and providing us feedback on the prototype versions of our translation tools.

References

[1] Joanne M. Atlee and John D. Gannon. State-based model checking of event-driven system requirements. In *Proceedings of the ACM SIGSOFT '91 Conference on Software for Critical Systems. Software Engineering Notes. Volume 16 Number 5*, 1991.

[2] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriam K. Rajamani. Automatic predicate abstraction of c programs. *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*, 36(5):203–213, May 2001.

[3] Tevfik Bultan, Richard A. Gerber, and William Pugh. Model-checking concurrent systems with unbounded integer variables: Symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems*, 21(4):747–789, July 1999.

[4] James C.Corbett. Constructing compact models of concurrent java programs. In *International Symposium on Software Testing and Analysis*, March 1998.

[5] W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J.D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.

[6] Yunja Choi, Sanjai Rayadurgam, and Mats Heimdahl. Automatic abstraction for model checking software systems with interrelated numeric constraints. In *Proceedings of the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE-9)*, pages 164–174, September 2001.

[7] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transaction on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[8] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[9] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proc. of the 22nd Int'l Conf. on Software Engineering*, pages 439–448, June 2000.

[10] J. Dingel and T. Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In *Computer Aided Verification*, pages 54–69, 1995.

[11] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *Proceedings of the Computer Aided Verification(CAV 1997)*, pages 72–83, 1997.

[12] O. Grumberg and D.E.Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[13] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[14] Nicolas Halbwachs, Fabienne Lagnier, and Chirstophe Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language lustre. *IEEE Transactions on Software Engineering*, pages 785–793, 1992.

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

[16] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, pages 366–381, 2000.

[17] Mats P.E. Heimdahl, Jeffrey M. Thompson, Barbara J. Czerny, and Duminda Wijesekera. Specifying and analyzing system level inter-component interfaces. Technical Report TR 98-030, University of Minnesota, Department of Computer Science, Minneapolis, MN, 1998.

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

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

[20] Z. Manna, M. Colon, B. Finkbeiner, H. Sipma, and T. Uribe. Abstraction and modular verification of infinite-state reactive systems. In *Requirements Targeting Software and Systems Engineering (RTSE), LNCS*. Springer Verlag, 1998.

[21] Kedar S. Namjoshi and Robert P. Kurshan. Syntatic program transformations for automatic abstraction. In *12th International Conference, CAV2000*, pages 435–449, July 2000.

[22] Formal Methods - Model Checking. Available at http://www-2.cs.cmu.edu/ modelcheck/.

[23] NuSMV: A New Symbolic Model Checking. Available at http://nusmv.irst.itc.it/.

[24] S. Owre, N. Shankar, and J.M. Rushby. *User Guide for the PVS Specification and Verification System*. Computer Science Laboratory; SRI International, Menlo Park, CA 94025, beta release edition, March 1993.

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

[26] Jeffrey Michael Thompson. *Structuring Formal State-Based Specifications for Reuse and the Development of Product Families*. PhD thesis, University of Minnesota, 2002.