Using PVS to prove Properties of Systems Modelled in a Synchronous Dataflow Language*

Sanjai Rayadurgam, Anjali Joshi, and Mats P.E. Heimdahl

Department of Computer Science and Engineering
University of Minnesota, Minneapolis
{rsanjai,ajoshi,heimdahl}@cs.umn.edu

Abstract. We report on our experience with using the PVS theorem prover as a verification tool for analyzing systems modelled in RSML^{-e} – a synchronous dataflow language. RSML^{-e} is a formal specification language particularly well-suited for specifying requirements of reactive systems. We advocate a *specification-centered approach* to system development, in which various development activities like prototyping, analysis, verification, testing, and code-generation are based on a formal model of the system requirements. To support the analysis and verification activities, we developed a translator from RSML^{-e} to PVS as part of our toolset. We used these tools to successfully verify properties of the mode logic of a flight-guidance system specified in RSML^{-e} by our industrial partner, Rockwell Collins Inc. The results from this exercise are encouraging. This paper describes our approach to formalizing RSML^{-e} in PVS and discusses briefly the strategies adopted in proving properties as well as some experiences.

1 Introduction

Software development for critical control systems, such as the software controlling aeronautics applications and medical devices, is a costly and time consuming process. Verification and validation of such systems must be an ongoing process throughout the development life-cycle. Currently, inspections and testing are the validation and verification methods used. We advocate that these methods be complemented with model checking and theorem proving. Also, other early life-cycle approaches like prototyping and specification simulation helps the analyst to evaluate and address poorly understood aspects of the system behavior. We advocate a specification-centered approach to development, in which a formal model of the system requirements is used to drive these life-cycle activities.

To realize a concrete instantiation of this approach, we have constructed a framework for developing tools to support specification-centered development using RSML^{-e} as the requirements specification language. RSML^{-e} [15] is a formal specification language particularly well-suited for specifying requirements of reactive systems. The NIMBUS toolset [14] provides the capability to execute

* This work has been partially supported by NASA contract NCC-01-001.

RSML^{-e} specifications. We have extended the analysis capabilities of the toolset by constructing translators to various verification tools such as NuSMV [12] and PVS [7]. Having the capacity to use different techniques like model-checking and theorem-proving to analyze the same RSML^{-e} model helps us leverage the unique advantages of each of these techniques. We have conducted case-studies using realistic industrial models to validate our approach.

In this report we present our formalization of RSML^{-e} in PVS. We have implemented a translator in the NIMBUS tool and used NIMBUS and PVS to verify various interesting properties of the mode logic of a realistic flight guidance system. Our motivation for the translation project and in general for using PVS as a verification tool was to help us prove classes of properties we could not prove using model-checking techniques. We also wanted to evaluate: (1) the feasibility of using a theorem prover as an *analysis back-end* to a specification tool, (2) the difficulty of constructing proofs, and (3) the scalability of the approach to industrial size systems.

In constructing translators to different verification tools our goals for the translation were driven by the specific capabilities of the tool and the expected user-interaction with the tools. Thus, when translating to the model-checker [5], we built in certain conservative abstractions that would make model-checking feasible by sacrificing some accuracy and expressiveness of the original RSML^{-e} specification. This was an acceptable trade-off since model-checking, when feasible, is completely automated and does not require any user interaction.

On the other hand, theorem-proving is essentially an interactive process. Thus, readability of the translated output and maintaining a close correspondence with the source specification were of importance. Further, there is no need to abstract away details in the source specification. Thus, the requirements for the translation were that it should fully capture the semantics of the source language in an elegant way producing readable PVS specifications.

Our formalization of RSML^{-e} in PVS is built around the concept of *objects as streams*, an idea similar to that of [2]. All entities in the specification – such as, variables, expressions and assignments – are viewed as state-indexed sequences of values. The specification, taken in totality, is considered as a set of constraints on the possible execution traces (*histories*) of the system. Verification, in this context, is checking whether the set of possible histories as constrained by the specification, satisfy a given predicate. Our formalization has the advantage of retaining both the structure and the semantics of the RSML^{-e} source specification in the translated PVS output. With some carefully chosen syntax, this makes the full power of the theorem prover available to the user at a level of representation that is in direct correspondence with the source specification. In our experience this has been of practical significance when constructing PVS proofs.

Our experience so far has been encouraging. Currently, the proof construction part is essentially a manual process. Even though the proofs are typically large, they are straightforward to construct. The complexity of the proofs does not seem to grow excessively with the increased complexity of the models. Where

the model checking efforts increase exponentially, our experiences indicate that the effort involved in constructing PVS proofs will exhibit a more *linear growth pattern*. We are in the process of empirically testing our hypothesis about the scalability of PVS proofs.

The rest of the paper is organized as follows. The next section briefly discusses the related efforts in this area. Section 3 provides an overview of the formal specification language RSML^{-e} and the PVS theorem prover. Section 4 describes our translation scheme in detail. Section 5 discusses our approach to proving properties. We then conclude the paper with a brief discussion in Section 6.

2 Related Work

We briefly discuss some of the related works in the area of using theorem proving for verifying reactive systems.

Owre *et al.* [13] discuss a systematic way to represent state-machine specifications of reactive systems in PVS, such as specifications written in SCR [6]. Extending that approach to RSML^{-e}, however, made reasoning with large systems a bit cumbersome. This was primarily due to the difficulty in understanding the mechanically translated PVS output and relating it to the original RSML^{-e} specification, a task that was often required during proof construction.

Bensalem *et al.* [2] discuss a methodology for proving control systems specified in Lustre using PVS. Their approach involves representing Lustre objects as streams in PVS, similar to the one that we describe here. They present a method for constructing provably correct control programs using Lustre and PVS in combination. An advantage of this approach is that property specification is not different from the specification of the system requirements.

TAME [1] is an interface for verifying properties of automata like, I/O automata, Lynch-Vaandrager timed automata and SCR. It provides a set of templates for specifying these automata and also a set of specializing strategies for reasoning about these automata in PVS. An advantage of this approach is that users can construct proofs using template strategies which are more meaningful and intuitive in the context of automata verification, without having to understand the underlying PVS steps. In our work, we only have a few hand-crafted specialized strategies that are used in reasoning about RSML^{-e} specifications. But this has been adequate to construct proofs of non-trivial properties on fairly complex models. We are currently working on constructing specialized strategies and investigating auto-generation of model-specific strategies to speed-up the proof construction process.

3 Framework

Figure 1 shows an overview of our verification framework. The user builds a behavioral model of the system in the fully formal and executable specification language RSML^{-e}. The specification is then fed to the Nimbus simulator which

checks that the specification is well formed and type correct. After the specification is checked, the user can translate the specification to the PVS or NuSMV input languages. The specification can then be analyzed for various properties

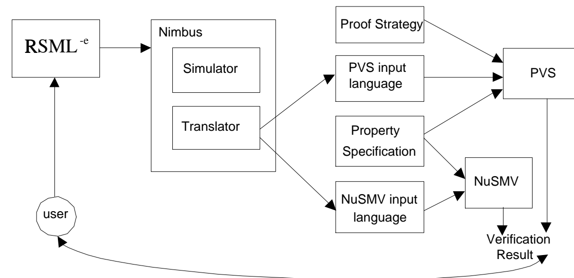

Fig. 1. Verification Framework.

using the theorem prover. The user can also input proof strategies to aid the proof process.

3.1 Flight Guidance System

A Flight Guidance System (FGS)[1] is a component of the overall Flight Control System (FCS). It compares the measured state of an aircraft (position, speed, and altitude) to the desired state and generates pitch and roll guidance commands to minimize the difference between the measured and desired state. The FGS can be broken down to mode logic, which determines which lateral and vertical modes of operation are active and armed at any given time, and the flight control laws that accept information about the aircraft's current and desired state and compute the pitch and roll guidance commands. We will be using a scaled down version of FGS as a running example in these discussions, but the equivalent properties to the examples in this paper have been proven on larger FGS models.

Figure 2 illustrates a graphical view of a FGS in the Nimbus environment. The figure shows the hierarchical and parallel state machines representing the different modes in the FGS. The arrows represent the possible transitions between states. The primary modes of interest in the FGS are the horizontal and vertical modes. The horizontal modes control the behavior of the aircraft about the longitudinal, or roll, axis, while the vertical modes control the behavior of the aircraft about the vertical, or pitch, axis. In addition, there are a number of auxiliary modes, such as half-bank mode, that control other aspects of the aircraft's behavior.

[1] We thank Dr. Steve Miller and Dr. Alan Tribble of Rockwell Collins Inc. for the information on flight control systems and for letting us use the RSML^{-e} models they have developed during our collaboration.

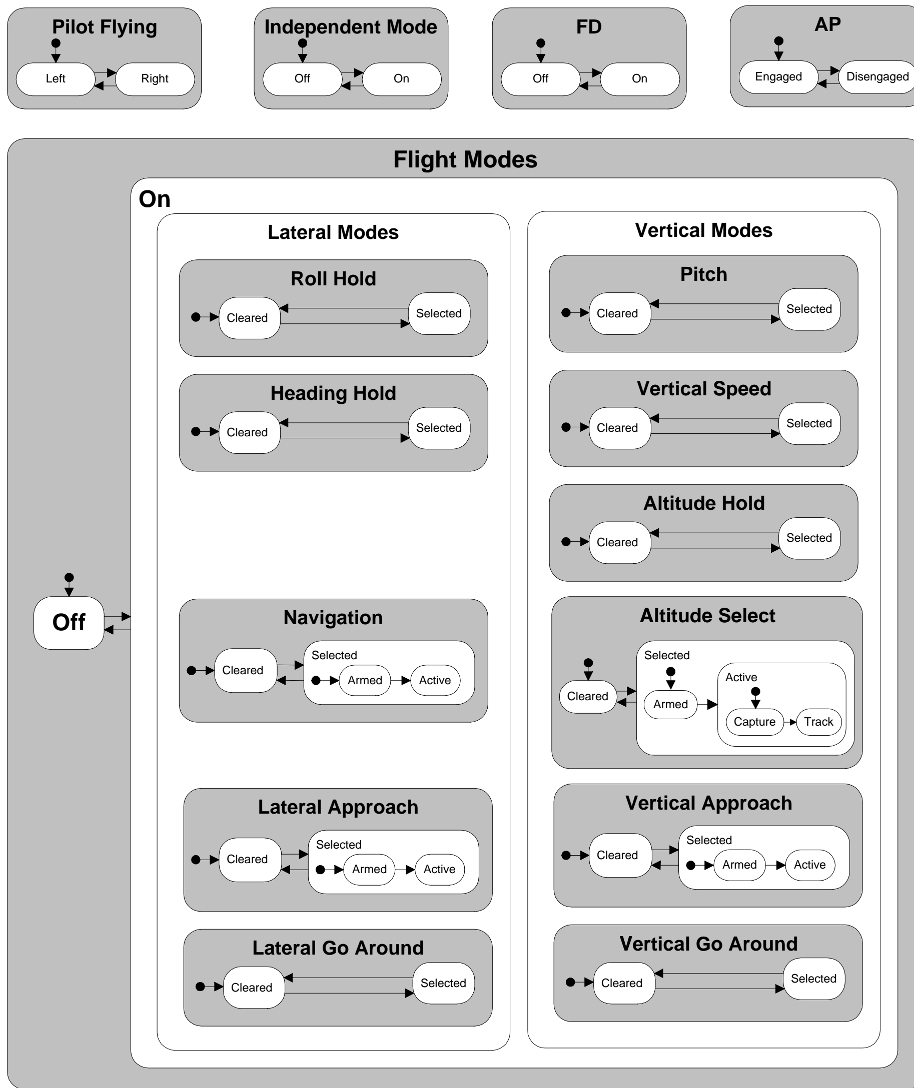

Fig. 2. Flight Guidance System

3.2 Overview of RSML^{-e}

RSML^{-e} stands for Requirements State Machine Language without Events. It is based on the Statecharts [8] like language Requirements State Machine Language (RSML) [11]. It is fully formal and a synchronous data-flow language without any internal broadcast events, which have been found to be error-prone [10].

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.

```
STATE_VARIABLE ROLL : Base_State
        PARENT          : Modes.On
        INITIAL_VALUE   : UNDEFINED
        CLASSIFICATION  : State
        TRANSITION UNDEFINED TO Cleared IF NOT Select_ROLL()
        TRANSITION UNDEFINED TO Selected IF Select_ROLL()
        TRANSITION Cleared TO Selected IF Select_ROLL()
        TRANSITION Selected TO Cleared IF Deselect_ROLL()
END STATE_VARIABLE

MACRO Select_ROLL() :
        TABLE
                Is_No_Nonbasic_Lateral_Mode_Active()    : T;
                Modes = On                              : T;
        END TABLE
END MACRO

MACRO Deselect_ROLL() :
        TABLE
                When_Nonbasic_Lateral_Mode_Activated()  : T *;
                When(Modes = Off)                       : * T;
        END TABLE
END MACRO
```

Fig. 3. A small portion of the FGS specification in RSML^{-e} .

Figure 3 shows a specification fragment of an RSML^{-e} specification of the Flight Guidance System[2]. The figure shows the definition of a state variable, `ROLL`. `ROLL` is the default lateral mode in the FGS mode logic. The state variable `ROLL` is declared as a child state of `Modes` and is active when the variable `Modes` has the value `On`—this notion of hierarchical variables provides the same abstractions and structuring mechanism as the AND and OR states in Statecharts , but the semantics is much simpler [15].

The conditions under which the state variable changes value are defined in the `TRANSITION` clauses in the definition. The condition tables are encoded in

[2] We use here the ASCII version of RSML^{-e} since it is much more compact than the more readable typeset version.

the macros, Select_ROLL and Deselect_ROLL. The use of macros not only improves the readability of the specifications but also helps localize errors and future changes. The conditions are represented in the AND-OR table format. 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.

Frequently we might need to refer to values of the variables at a certain point in the variable *history*. RSML^{-e} provides a construct for doing this, as shown in the following example.

```
MACRO Were_Modes_Off() :
        PREV_STEP(Modes) = Off
END MACRO
```

In the above example, PREV_STEP(Modes) refers to the previous value of the state variable Modes.

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 [2]. Data-flow semantics removes complex issues caused by internal events, such as infinite triggering events or analysis of micro-steps [4], 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 modeling of this uncertainty by providing the concept of *Undefinedness*. One can explicitly specify the initial value of variables at startup to be *Undefined*, such as ROLL=UNDEFINED in Figure 3. 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 both explicit and implicit *Undefinedness*.

3.3 Properties of interest for theorem proving

Most of the FGS properties could be expressed as state invariants for verification. State invariants are suitable for model checking and indeed around 290 FGS properties have been successfully model checked for the largest FGS model [5]. However, there were also some other types of interesting properties, like the FGS mode confusion properties [3], some of which cannot be model checked. As an example, consider the following property:

> Any two states that do not have the same modes, have different mode annunciations.

This property compares two arbitrary states, with no particular values specified for modes or mode annunciations, other than saying that the values are either same or different [9]. This property cannot be expressed in the temporal logics used by conventional model-checkers. Proving such mode confusion properties was a motivation for the current work in exploring the application of a theorem prover like PVS.

3.4 Overview of PVS

PVS [7] (Prototype Verification System) provides an environment for effective proof construction in addition to writing specifications. Its input language is based on simply-typed higher-order logic with function, record and product types and recursive type definitions. The language provides a powerful mechanism to specify and use sub-types. The powerful type system means that type-checking a PVS specification is in general an undecidable problem. Type-checking could require guidance to the theorem prover from the user in dismissing type correctness conditions. PVS specifications are organized into theories that can be parameterized. The primitive proof steps are composed of efficient decision procedures, rewriting rules and BDD based propositional simplifications.

4 Translating from RSML^{-e} to PVS

We considered two competing approaches for representing RSML^{-e} specifications in PVS.

The first approach is to view the *state-space as a cross product of the domains* of system variables. The specification is viewed as a collection of constraints determining the set of possible initial states and the set of possible transitions between states. Then, the transitive closure of the initial states under the transition relation constitutes the reachable state-space of the system. The system will satisfy a certain property of interest if it can be established that every reachable state satisfies this property. This view is usually adopted when one is verifying state-based specifications using model-checkers like SMV or the μ-calculus model-checker of PVS. Owre *et al.* [13] discuss such an approach to translate requirement specifications written in SCR to PVS.

The second approach is to consider *state as a point of observation of certain quantities of interest* in the system. The system variables represent quantities of interest, i.e., they are mappings from states (the observation points) to values of those quantities (at those observation points). When the system responds to changes in its environment, it moves to a new observation point, i.e., to a new state. So each state has an associated (finite) *history* of observations up to that point. In this view, the system specification is a set of constraints on the histories of observations at each state. If we think of constraints as Boolean valued quantities constructed using system variables, then the specification lists a set of such quantities that are given to be true in every state. Properties of interest that one wants to prove are also similar to constraints but one has to

establish that these are true. Bensalem *et al.* [2] adopt such an approach for proving properties of control system specified in LUSTRE using PVS.

In an earlier version of our translation we adopted the former approach to translate RSML^{-e} specifications to PVS. However, we found that it was difficult to construct proof of properties in PVS for large systems using such an approach. Part of the difficulty arose from the fact that one had to carry around the complete state construct in proofs, even though, much of the reasoning and proof steps involved only a few variables at any given time. Also, the translated output was quite difficult to comprehend. The latter approach, which we adopted subsequently, overcomes these shortcomings.

In the current approach, objects defined in RSML^{-e} are treated as sequences of values over states in PVS, also called *streams*. Operations over values are uniformly lifted to operations over streams by applying the operation to values at each state. The resulting translation to PVS retains a close correspondence with the original RSML^{-e} specification making it easy to understand and follow. In the next two subsections we discuss the translation scheme in detail.

4.1 Translation Foundation

As the first step to translation, we defined a library *rsmlne.pvs* containing definitions for various constructs of the RSML^{-e} language in PVS. These RSML^{-e} constructs include the basic types, operations to lift RSML^{-e} objects to streams, RSML^{-e} specific operations, and so on. This PVS library will then be imported into every translated RSML^{-e} specification, so that the basic definitions can be reused across all specifications. Due to space constraints, we present only the most relevant aspects of the translation scheme here.

Undefined and Defined Values: In RSML^{-e} variables may be *undefined* in certain configurations (global states) of the system. To capture this notion, we uniformly lift all RSML^{-e} types to include a null element. Defined values are accessed using a C-like address/contents (`&/*`) syntax:

```
rType[T: TYPE]: DATATYPE
 BEGIN
  null    :  undef?
  &(*: T) :  def?
 END rType
```

A generic function `ext` extends operation on T-values to null-extended-T-values by applying the operation to the contents when the value is defined and otherwise returning null.

States and History States: As explained earlier, states are just points of observations of quantities of interest in the system. The only properties that we require of states are that: 1) There is some starting point for observations - initial state, and 2) There is a unique history for each state - previous state. In our theory, we define State as a natural number: the initial state being zero and n being the predecessor of n + 1, and zero being the predecessor of itself.

```
State: TYPE = nat
init: State = 0
```

RSML^{-e} specifications may use certain types of history operations on objects like variables and interfaces. These operations may access values of variables at certain points in their history (for example, when the value changed the second previous time). To uniformly translate such expressions, we define a single history operation on States called `last`. This is a higher order recursive function, which results in a state transformer, i.e., the result is a mapping from a state to a (previous) state. It returns the `z`th `last` state at which the `take` function was true.

```
prev(s: State): State = pred(s)
history?(s: State)(x: State):bool= x <= s
Filter: TYPE = [State -> bool]
last(take: Filter, z: posnat)(s:State): RECURSIVE (history?(s)) =
    IF s = init OR (z = 1 AND take(s)) THEN s
    ELSE last(take, IF take(s) THEN z - 1 ELSE z ENDIF) (prev(s))
    ENDIF
  MEASURE s
```

Objects as Streams: RSML^{-e} objects are implemented as streams that map State to (null-extended) values:

```
Object: TYPE = [State -> rType[T]]
```

In certain contexts, objects may have to be constrained to be defined (or undefined) at every state. The predicates `defined?` and `undefined?` are used for this purpose. The object `UNDEFINED` returns *null* at each state. The `L` operator lifts the RSML^{-e} constants to streams.

```
L(x: T): (constant?[T]) = LAMBDA s: &(x)
```

The RSML^{-e} entities, state variables and input variables and the return types of functions and macros, are simply objects of the appropriate type; constants are constant objects of their respective types; and, conditions (such as those guarding assignments to variables) are defined Boolean objects.

```
rSTVAR__:  TYPE = Object[T]
rINVAR__:  TYPE = Object[T]
rFUNCT__:  TYPE = Object[T]
rMACRO__:  TYPE = Object[bool]
rFIELD__:  TYPE = (defined?[T])
rCONST__:  TYPE = (constant?[T])
rCOND__ :  TYPE = (defined?[bool])
```

Messages in RSML^{-e} are data received and sent by the system at the interfaces and such data are always defined. We represent Messages from RSML^{-e} as records in PVS, with fields having `rFIELD__[T]` type for the appropriate type `T`.

RSML^{-e} Basic Types: The RSML^{-e} types, `BOOLEAN`, `REAL`, `INTEGER` have equivalent types in PVS. RSML^{-e} type `TIME` is simply of type nonnegative real in PVS. `TIME` is an intrinsic object in RSML^{-e} that is always defined and is monotonically increasing with respect to State.

RSML^{-e} Specific Operations: Here we define a few RSML^{-e} specific operations on objects. Equality comparison is a safe-operation even if one or both of its operands are Undefined. That is its result will always be defined. We use the symbolic operator `==` to distinguish it from the normal equality operator `=`.

```
==(obj1, obj2: Object[T]): (defined?[bool]) =
      LAMBDA (s: State): &(obj1(s) = obj2(s))
```

The unary operator `PREV` yields an object that gives the value of its operand in the previous state:

```
PREV(obj: Object[T]): Object[T] = LAMBDA (s: State): obj(prev(s))
```

We require a binary `BECAME?` operation on objects, with the intuitive meaning "did the first object's value change to that of the second object in this state?". Also required is a unary `CHANGED?` that simply checks if the object's value changed in this state.

```
BECAME?(obj1, obj2: Object[T]): (defined?[bool]) =
      LAMBDA s: &(obj1(s) /= PREV(obj1)(s) AND obj1(s) = obj2(s));

CHANGED?(obj: Object[T]): (defined?[bool]) =
      LAMBDA (s: State): &(obj(s) /= PREV(obj)(s));
```

With this formulation, one could express SCR style operators like @`T(expr)`, @`F(expr)` and @`C(expr)` as `BECAME?(expr, true)`, `BECAME?(expr, false)` and `CHANGED?(expr)` respectively. Also note that these could be used to determine history states in a general fashion, such as, the nth last time variable A changed. The function `PREV_STATE` takes an object representing a condition and the step count to compute the appropriate history state:

```
PREV_STATE(c: Object[bool], z: posnat):[s:State -> (history?(s))] =
      last(* o (c == L(TRUE)), z)
```

A Lustre style followed-by operator (->) is useful in expressing initial state values for RSML^{-e} entities. We overload the `ANDTHEN` infix operator in PVS for this purpose, which conveys the intuitive meaning of followed-by:

```
ANDTHEN(obj1, obj2: Object[T]): Object[T] =
      LAMBDA (s: State): IF s = init THEN obj1(s) ELSE obj2(s) ENDIF
```

Similarly, we also overload the `WHEN` infix operator in PVS to express parent state constraint. The expression `(A WHEN B)` will have the value of `A` when the condition `B` is true in a state and otherwise be undefined:

```
WHEN(obj: Object[T], p: (defined?[bool])): Object[T] =
      LAMBDA (s: State): IF *(p(s)) THEN obj(s) ELSE UNDEFINED(s) ENDIF
```

Guards and Guarded Expressions: Computation in RSML^{-e} specifications is expressed in terms of guarded assignments to variables. An assignment is triggered if its corresponding guard evaluates to true. Thus, for *consistency* (or to avoid non-determinism), the guards of different assignments of a variable must be disjoint at each state. Also, for *completeness*, the disjunction of the

guards of all assignments of a variable must be a tautology. The construct, `COND ... ENDCOND` in PVS, is typically used to capture such guarded expressions. It is equivalent to a series of if-then-else expressions, except that it generates disjointedness and completeness constraints as type-correctness obligations to be proved by the user. One could lift this construct state-wise to streams (and thus to RSML^{-e} objects) by evaluating the guard and the expressions at each state. The RSML^{-e} library for PVS defines operators `[| |]`, `>>`, `/\` and `ELSE?`, such that,

```
[|  ... >> ... /\ ... /\ ELSE? >> ... |]
```

is equivalent to lifting,

```
COND ... -> ... ,  ... ,  ELSE  -> ... ENDCOND
```

to RSML^{-e} objects, state-wise. The translator can be set to generate default `ELSE?` cases that just stutter previous state values. This is useful when the specification is written assuming the implicit behavior of "no change in value when none of the guards are true".

4.2 RSML^{-e} to PVS translation

On the basis of the translation foundation discussed above, we will now illustrate the actual translation of various RSML^{-e} constructs. The *rsmlne.pvs* library will be imported into the translated PVS specification. The basic construct in RSML^{-e} is the variable and the transition relation defined on the variable—the translation of these constructs is discussed in detail below.

Type definitions: Basic RSML^{-e} types are defined in the *rsmlne.pvs* library. RSML^{-e} enumerated types are defined in a straightforward way in PVS.

```
TYPE_DEF Base_State   {Cleared, Selected}
```

translates to

```
Base_State: TYPE =    {Cleared, Selected}
```

Variable Declarations: Input variable declarations create equivalent PVS definitions for the type, if necessary. The expected_min and expected_max specifications, if they exist, are not declared, but just translated as constants wherever those are used. The unit and classification definitions are ignored since they are primarily present for documentation purposes.

```
IN_VARIABLE FD_Switch: Switch
            INITIAL_VALUE : UNDEFINED
            CLASSIFICATION: MONITORED
        END IN_VARIABLE
```

translates to

```
FD_Switch:  rINVAR__[Switch]
```

The state variable declarations are handled in a similar way as the input variables. The declaration for the state variable `ROLL` from figure 3 translates to

```
ROLL:  rSTVAR__[Base_State]
```

Functions and Macros: Functions and macros are both defined as functions, with macros being functions returning Boolean values. The macro `Deselect_ROLL` from figure 3 translates to

```
Deselect_ROLL: rFUNCT__[BOOLEAN] =
       When_Nonbasic_Lateral_Mode_Activated
       OR
       BECAME?((Modes == L(Off)), L(TRUE))
```

State Variable Assignments: The bulk of the computation in an RSML$^{\text{-}e}$ specification is in the assignments and their guard conditions expressed as AND/OR tables. While there could be no cycles in the dependency among variable values at a given state in a correct RSML$^{\text{-}e}$ specification, assignment expressions and guards may frequently refer to one or more history state values of any number of variables. Thus, variable histories are defined by a set of mutually recursive equations. This mutual recursion cannot be directly represented in PVS. This problem is also addressed in the [2] as the *feedback loop problem*. To handle the mutual recursion, we split the definitions for variables into three parts:

- Declaration of the variable
- Defining equation for the variable
- Assertion that the variable is equal to the value given by its defining equation.

The declarations of variables is explained earlier. For the rest of the definition: First the guard conditions are translated to individual condition objects. While this is not necessary, it makes the translator output readable and easy to follow. If the assignment is given as a transition from one state to another, it is internally rewritten to an assignment form where the guard includes the previous state as a constraint. As an example, the first guard condition for the `ROLL` state variable from figure 3,

```
TRANSITION UNDEFINED TO Cleared IF NOT Select_ROLL()
```

translates to

```
ROLL__T1:  rCOND =
        (NOT Select_ROLL)
        AND
        (PREV(ROLL) == UNDEFINED)
```

The transition relation needs to consider the hierarchical relationship between variables. The transition relation for child variables needs to check if the parent variable has the right value. If the parent has the right value, the child variable is relevant and its transition relation is evaluated normally. If the parent variable has the wrong value, then the child will be *undefined*. This is captured using the `WHEN` operator.

The complete definition of the `ROLL` state variable is translated as,

```
ROLL__DEF: rFUNCT__[Base_State] =
    (UNDEFINED
     ANDTHEN ([|   ROLL__T1 >>  L(Cleared)
              /\  ROLL__T2 >>  L(Selected)
              /\  ROLL__T3 >>  L(Selected)
              /\  ROLL__T4 >>  L(Cleared)
              /\  ELSE?  >>  PREV(ROLL)
             |]
              WHEN  (Modes == L(On))))

ROLL__DECL: AXIOM ROLL = ROLL__DEF
```

Finally, an axiom is generated to assert that the variable is equivalent to the value given by its defining equation. However, this axiom could seldom be used as an auto-rewrite unconditionally, for this could easily cause rewrite loops. So, two additional conditional equations - one for the initial state and one for the non-initial states - are generated, which could be used as auto-rewrites in proof strategies:

```
s: VAR State
ROLL__INIT: AXIOM  (s = init)  IMPLIES  ROLL(s) = ROLL__DEF(s)
ROLL__NEXT: AXIOM  (s /= init) IMPLIES  ROLL(s) = ROLL__DEF(s)
```

While the indiscriminate use of axioms could lead to inconsistent specifications, the type-correctness of RSML^{-e} specifications is sufficient to guarantee that this is not the case with the translated PVS output. In particular, RSML^{-e} language disallows cyclic dependency among variables.

The input variable assignments are handled similar to the state variable assignments. Although input variables have a different syntax and their assignments appear under handlers for input interfaces, the PVS translation produced is similar to that of state variables. In other words, input variables are treated very much like top-level state variables in the PVS interpretation.

Messages, Interface Declarations and Handlers: The translator can be set to skip interfaces altogether when one is interested in reasoning about the specification independent of input and output. In that case the input variables are all left unconstrained so that they may assume any value at each step.

Messages are declared to be of a record type in PVS. The interfaces are declared to be constants of the type of the messages handled. The interface message separation times are translated to constant objects wherever they are used. The input interface handlers define the values of the input variables and thus provide the transition conditions for the input variable assignments.

One-Input Interface Assumption: In RSML^{-e} step computation is assumed to take place when and only when an input is received by one of the input receivers. Also, it is assumed that two input receivers do not receive messages from the environment at the same instant and that input values do not change before computation is completed. Thus, the trigger for computation of a step is receipt

of a single message. It is implicitly assumed that there is a system clock interface, which periodically receives clock ticks, so that even if there are no other receivers, computation still proceeds. To capture the one-input assumption when there is more than one receiver, a system input variable INPUT? is declared, whose possible range of values are the different input receivers in the specification. Its value at each state is understood to be the receiver that triggered the step computation for that state.

5 Proving Properties

Currently, the proof construction is essentially a manual process. The most common properties we encountered during the verification of the flight-guidance mode logic were *state invariants* (p is true is always true) or *transition invariants* (if p is true in the current state, p will be true in the next state). While proofs for transition invariants begin with a CASE split for the *init* and *next* states, those for state invariants begin with instantiation of a simple induction schema over states. After the first steps, the proofs of the subgoals, follow a similar pattern in both types of proofs, the details of which follow. The subgoals that one has to address typically could involve the current (or previous) state either in the *init* or the *next* state configuration. Most proofs do not require reasoning beyond one previous state in the history. However, RSML^{-e} allows the use of state history of any bounded length, and, therefore, there could be specifications for which proofs may require reasoning beyond one history state. Below, we discuss briefly how to proceed with a proof after we have instantiated an induction schema, so that we have two subgoals to dismiss: one to show that the property holds in the initial state and one to show that is holds in the next state.

Example Proof: Our motivation for using the theorem proving approach were the *mode confusion* properties. Though they are really interesting properties, they have rather involved proofs. For the purpose of illustration, we will consider here a simple *state invariant* that we may wish to verify on a toy model of the Flight Guidance System. Though this property is clearly suited for model checking, we use it here as a simple example to explain the general proof process. The proofs for the mode confusion properties [9] follow a similar pattern.

```
At_Least_One_Lateral_Mode_Active : THEOREM
    verify(Mode_Annunciations_On IMPLIES
                      (Is_ROLL_Selected OR Is_HDG_Selected))
```

Informally, the property states, as the name implies, that whenever modes are turned on, at least one lateral mode is active. In more realistic models, there would be several of those modes, some classified as lateral and some classified as vertical. The proof of this property for our much larger models, follows a similar sequence of steps.

Invocation of basic auto-rewrite strategies (described later) followed by simplification, reduces the goal to:

```
  |-------
{1}   FORALL (s: State):
        IMPLIES(*(Modes(s)) = On,
                OR((ROLL(s) = &(Selected)),(HDG(s) = &(Selected))))
```

Since this is a state invariant, we decide to induct on state `s`. This yields two subgoals: 1) `s` is an *init state*, and 2) `s` is a *next state*. The *init* branch can be dismissed trivially by invoking Modes__INIT which asserts that Modes = Off in the init state. Since the left-hand-side of the implication is false, the subgoal will be immediately dismissed.

The *next* state branch, after skolemization and simplification, becomes:

```
[-1]  IMPLIES(*(Modes(j!1)) = On,
              OR((ROLL(j!1) = &(Selected)),(HDG(j!1) = &(Selected))))
{-2}  *(Modes(1 + j!1)) = On
  |-------
{1}   (ROLL(1 + j!1) = &(Selected))
{2}   (HDG(1 + j!1) = &(Selected))
```

Note that $[-1]$ formula in the antecedent is the induction hypothesis. State `(j!1)` is the previous state and `(1 + j!1)` is the current state[3] in the induction process. Proceeding with the proof, we may now instantiate the transition relation for the `ROLL` (or, symmetrically, `HDG`) state variable assignments in the current state and simplify to obtain:

```
{-1}  ROLL(1 + j!1) =
       IF *(ROLL__T1(1 + j!1))  THEN &(Cleared)
       ELSE IF *(ROLL__T2(1 + j!1))  THEN &(Selected)
            ELSE IF *(ROLL__T3(1 + j!1))  THEN &(Selected)
                 ELSE IF *(ROLL__T4(1 + j!1))  THEN &(Cleared)
                      ELSE ROLL(j!1)
                      ENDIF
                 ENDIF
            ENDIF
       ENDIF
[-2]  IMPLIES(*(Modes(j!1)) = On,
              OR((ROLL(j!1) = &(Selected)),(HDG(j!1) = &(Selected))))
[-3]  *(Modes(1 + j!1)) = On
  |-------
[1]   (ROLL(1 + j!1) = &(Selected))
[2]   (HDG(1 + j!1) = &(Selected))
```

Now, the value of `ROLL` in the current state depends on the guard conditions satisfied. Dismissing the conditions one by one, CASE splitting as required, would result in a sub-goal like the following:

[3] For clarity of presentation we talk of previous/current states, instead of current/next states to avoid confusing with init/next states

```
[-1]  *(When_HDG_Switch_Pressed(1 + j!1))
{-2}  *(ROLL(1 + j!1)) = Cleared
[-3]  *(Modes(1 + j!1)) = On
  |-------
[1]   HDG(j!1) = &(Selected)
[2]   *(ROLL(j!1)) = Cleared
[3]   ROLL(j!1) = null
[4]   (ROLL(1 + j!1) = &(Selected))
[5]   (HDG(1 + j!1) = &(Selected))
```

Instantiation of the transition relation on `ROLL` led to the value of `Cleared` for the variable in the current state. At this point, it is clear that more information from the specification is necessary to proceed further with proof construction: we have not yet reasoned with the value of `HDG` variable in the current state. We, therefore, instantiate the transition relation for `HDG` in current state and introduce it into the sequent. Dismissing the various guard conditions for `HDG__NEXT` and further simplification of the consequent formulas, yields `HDG = Selected` for the current state, which is one of the consequents. This completes the outline of the proof of the invariant.

As mentioned earlier, the most common properties we encountered during the verification of the FGS were state and transition invariants. The proofs were fairly straightforward, though long. Most proofs followed a structure similar to the one explained above, where the analysis faces with two cases:

init State Invoke the *StateVar*__INIT condition to dismiss the proof branch. These sub-goals are trivial to dismiss.

next State Invoke the *StateVar*__NEXT transition condition on one of the State variables involved in the property. Since the transition condition is composed of *COND* statement, each branch, corresponding to each transition would have to be dismissed to obtain the value assigned for the state variable in the next state. If after simplification of the transition condition, the proof is not yet complete, we may have to deal with one of the following cases:

1. A subgoal is reached, whose consequent is provable but requires additional information to prove it. In this case, it may be necessary to invoke new initial/transition conditions on some other relevant state variables on which the value of the present state variable depends. We then repeat the above process.
2. A subgoal is reached, in which one of the newly introduced antecedents is false. This may again require introduction of more information into the proof branch through the initial/transition conditions to discharge the subgoal by contradiction.
3. A subgoal is reached, which is unprovable. This would typically point to a scenario in which the property being analyzed is false. The counter-example, could typically be easily gleaned from the formulas in the sequent. However, some familiarity with the system being verified may be necessary to determine that a sub-goal is unprovable.

The `At_Least_One_Lateral_Mode_Active` example proof described above is for a scaled down version of the FGS. This version of FGS is about 900 lines of PVS code, when translated from the RSML^{-e} specification. The complete proof is 93 proof steps and runs in approximately 6 seconds on a 1.5 GHz Linux workstation with 1.5 GB main memory. The largest FGS model that we have worked on is about 3900 lines of translated PVS code. In this version of FGS, there are five lateral modes instead of the two in our example property. The `At_Least_One_Lateral_Mode_Active` proof for this version of FGS is 380 proof steps and runs in approximately 40 seconds on the same machine. We also constructed around six elaborate proofs analyzing the mode logic of the FGS. One of those proofs, which is especially interesting, could not be verified using model checking techniques.

One of the authors was involved in constructing these proofs manually. The author had no prior experience in theorem proving before starting this exercise. An interesting observation from this exercise is that the effort involved in constructing these proofs, although large, increases roughly linearly with the size of the model. The experience so far has been very encouraging as we were successful in constructing non-trivial proofs of useful properties of a critical system model used in the avionics industry. We are cautiously optimistic that the theorem proving approach may well scale up to much larger systems than what we can handle using model-checking techniques.

6 Discussion

In our experience, the proofs we dealt with, have been typically *long, but straightforward to understand*. To make the translation specific details transparent to the user and increase readability of the sub-goals, we invoke the RSML^{-e} specific definitions from the library file as lazy, eager or macro auto-rewrites in PVS. Those are automatically brought in whenever the `ASSERT` primitive is used in the proof. To further reduce the tediousness of constructing proofs, we have attempted to construct certain non-trivial strategies that are specific to proofs of properties for RSML^{-e} models, as well as, generating model-specific strategies along with the translation. As a first step, we identified simple patterns of rule invocations and encoded these patterns as strategies. For example, a simple strategy `EXPAND_SIMP` is frequently used in these proofs. This strategy expands certain definitions and simplifies the result using a few other rewrite rules and lemmas. Simple strategies, such as this, have helped greatly in reducing the length of the proofs and remove the intermediate clutter while rendering the proof more readable. Since extensive automation is the goal of all our analysis work, the next step is to pursue construction of more powerful strategies that are both language and model specific. In the example proof discussed earlier, it is rather straightforward to determine that one has to introduce `ROLL` and `HDG` transition relations to complete the proof. But generating this automatically as a strategy from the specification is rather involved. It requires identifying patterns in the sub-goal and invoking the right set of rules and lemmas, while

at the same time, providing a fine-grained control to the user in choosing the proof steps to apply. Experience from our preliminary work in this area seems to suggest that such automated strategy generation is difficult. Manual proof construction provides a certain level of flexibility that lets the user determine the level of detail to which the specification must be drilled down during proof construction on a per subgoal basis. This flexibility has been critical for keeping the proofs readable and manageable. Construction of more powerful strategies seem to trade-off some of this flexibility. Our current efforts are directed towards finding the right balance between the two for typical proofs for RSML^{-e} models.

In conclusion, in this paper we presented a method for formalizing a synchronous dataflow language in PVS, based on which a mechanical translator was implemented. A salient feature of the translation scheme is that it reflects the structure and the semantics of the source specification, which has been useful in proof construction. We have been successful in verifying non-trivial properties of the mode logic in a flight-guidance system. Some of these properties could not be model-checked. We also observed that the proof complexity did not grow exponentially with the size of the model.

References

1. Myla Archer, Constance Heitmeyer, and Steve Sims. TAME: A PVS interface to simplify proofs for automata models. In *User Interfaces for Theorem Provers*, 1998.
2. S. Bensalem, P. Caspi, C. Parent-Vigouroux, and C. Dumas. A methodology for proving control systems with Lustre and PVS. In *Proceedings of the Seventh Working Conference on Dependable Computing for Critical Applications (DCCA 7)*, 1999.
3. Ricky W. Butler, Steven P. Miller, James N. Potts, and Victor A. Carreno. A formal methods approach to the analysis of mode confusion. In *17 th AIAA/IEEE Digital Avionics Systems Conference*, October 1998.
4. 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.
5. Yunja Choi and Mats Heimdahl. Model checking RSML^{-e} requirements. In *Proceedings of the 7th IEEE/IEICE International Symposium on High Assurance Systems Engineering*, October 2002.
6. P. Clements. *Software Cost Reduction through Disciplined Design*. 1984 Naval Research Laboratory Review, Washington D.C., 1985. Available as National Technical Information Service order number AD-A1590000, pp. 79-87, July 1985.
7. J. Crow, S. Owre, J. Rushby, et al. A tutorial introduction to PVS. In *WIFT 95: Workshop on Industrial-Strength Formal Specification Techniques*, 1995.
8. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
9. Anjali Joshi, Steve P. Miller, and Mats P.E. Heimdahl. Mode confusion analysis of a flight guidance system using formal methods. In *To appear in Digital Avionics Systems Conference*, 2003.
10. Nancy G. Leveson, Mats P.E. Heimdahl, and Jon Damon Reese. Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the

Future. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, volume 1687 of *LNCS*, pages 127–145, September 1999.
11. 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.
12. NuSMV: A New Symbolic Model Checking. Available at http://nusmv.irst.itc.it/.
13. S. Owre, J. Rushby, and N. Shankar. Analyzing tabular and state-transition requirements specifications in PVS. Technical Report SRI-CSL-95-12, SRI International, June 1995.
14. Jeffrey M. Thompson and Mats P.E. Heimdahl. An integrated development environment prototyping safety critical systems. In *Tenth IEEE International Workshop on Rapid System Prototyping (RSP) 99*, pages 172–177, June 1999.
15. Michael W. Whalen. A formal semantics for RSML^{-e}. Master's thesis, University of Minnesota, May 2000.