Requirements Analysis of Real-Time Control Systems using PVS

Bruno Dutertre   Victoria Stavridou

bruno@dcs.qmw.ac.uk, victoria@dcs.qmw.ac.uk
Department of Computer Science,
Queen Mary and Westfield College,
University of London

Abstract

This paper presents a practical application of the PVS theorem prover involving requirements analysis of real-time control systems. This work was conducted within the SafeFM project and relied on a real world avionics case study. We show how PVS was used to formalize the software requirements for the system and to verify safety-related properties. We also present the main result of the experiment. We give an overview of PVS libraries which were developed after the case study experiment and are intended to facilitate the specification and verification of similar systems.

1 Introduction

The SafeFM project[1] investigated the practical application of formal methods to the development and assessment of high integrity systems [17]. Within the project, we investigated the use of formal methods and theorem proving in requirement analysis of real-time control systems. A major part of this work involved an experiment applying the PVS theorem prover to the analysis of an avionics control system. The case study was a substantially complex example based on an existing system developed by GEC-Marconi, one of our SafeFM partners.

This paper presents the main results of SafeFM in the domain of formal requirement specification and analysis. Section 2 gives an overview of the verification method. Section 3 presents the formal notations we used and shows how PVS can provide mechanical support to formal requirements analysis. Section 4 is an overview of the case study experiment; it describes the successive phases of the formalization and verification using PVS, and presents the main experimental results. Section 5 describes further work dealing with one of the limitations of PVS identified during the case study: the lack of general purpose libraries.

[1]This work was partially funded under the UK Department of Trade and Industry SafeIT programme by EPSRC Grant No GR/H11471 under DTI Project No IED/1/9013.

2 Methodology

2.1 Applications Considered

SafeFM focused on a specific class of high-integrity systems, namely digital controllers. In particular, we were interested in medium size applications encountered in avionics such as air-data computers or store managers. These applications have several important characteristics:

- They are usually fault tolerant systems with a redundant architecture. Because of the need for high reliability, avionics controllers incorporate multiple processors so that hardware failures can be tolerated.
- They are real-time applications. Digital controllers interact with an active environment which imposes timing constraints. For example, input signals have to be sampled and processed at a sufficient rate for the system to maintain an accurate image of its environment. Commands have to be produced at the right time and may have to accommodate various externally defined mechanical constraints.
- They are hybrid systems. They may receive input both from discrete or from analogous sources and have to implement complex control laws which mix logical and numerical computations.

All these characteristics are related to a major element of all control applications: the system under control. Real time digital controllers cannot be understood and analyzed without taking into consideration the controlled system, its behavior, and its properties.

2.2 Method of Analysis

Our primary objective within SafeFM was to design a methodology for the formal analysis of software requirements for real-time digital controllers. Our investigation was guided by the SafeFM case study and we started from the following point of view:

- The control system consists of various processing units and the general architecture of the system (processors, communication links, interface, etc.) is given.
- The requirements describe functional modules to be implemented by each of the processors. Typically, each module defines a control function or some other task such as failure detection or monitoring. The requirements may include real-time and temporal constraints as well as purely functional aspects.

In order to get confidence in the validity of requirements, we want first to check that the description of each module is internally consistent. This assumes that the requirements can be specified formally and that various consistency checks can be performed on the formal specification. Verification might include type checking, the detection of out-of-range values, or the proof of general semantic properties. The preservation of invariants in state-based formalisms such as B[1] or VDM[10] is a typical example of such semantic verification. Although it is not exactly a consistency property, checking total coverage of the input domain is another important example of semantic validation [9, 15].

The class of applications we consider are often safety critical or safety related. In this context, checking only the internal consistency of functional requirements is not sufficient. We also need to be able to verify that critical properties are satisfied. Such properties are global constraints on the behavior of the system under control. Verifying that a system satisfies these high-level properties requires more than knowing the functional requirements of each module. Additional information, such as the architecture of the controller and a model of the system under control, is necessary.

In summary, we assume that specifications for real-time control applications can be structured into three broad classes: the functional requirements, a list of assumptions about the controlled system, and the critical properties to be verified. Coherence can be demonstrated by applying various consistency checks to individual functional modules and by proving that the critical properties are satisfied by the functional requirements.

3 Requirements Analysis with PVS

In order to perform the different kinds of verification mentioned above, some form of mechanical support is necessary. The tool must provide a rich formal notation able to cover the various classes of requirements of real-time control systems. It must also support reasoning about numerical as well as logical properties.

PVS was the tool chosen for SafeFM. PVS offers both a very expressive specification language based on higher order logic and a powerful theorem prover. A description of the system and examples of applications of PVS can be found at the PVS world-wide web site[2] and elsewhere [4, 13, 14, 16].

PVS is a general system and we had to define a specification approach for real-time systems requirements. We used a straightforward and easy to implement approach with explicit time. Time is modeled by the non-negative real numbers, time varying quantities are manipulated explicitly as functions of time, and temporal properties are written using explicit time indices.

There are two kinds of time dependent variables. The first kind is used to specify the requirements of discrete components; functions are defined on a subset of the time domain which represents a clock. The second is total function of time and models the continuous variables of an application.

3.1 Functional Requirements

In order to specify functional requirements, we used a data flow approach inspired from the synchronous languages LUSTRE and SIGNAL [8, 11]. Every module is assumed to be activated at regular intervals by a clock of known frequency. In PVS, such clocks are defined as follows:

```
clocks[K:posreal]: THEORY
  BEGIN

  IMPORTING time
```

[2] http://www.csl.sri.com/pvs.html

```
t: VAR time
n: VAR nat

clock: TYPE =
   {t | EXISTS n : t = n * K}.
```

A clock is a parameterized subtype of `time`, characterized by a positive real K – the period of the clock. An element x is of type `clock[K]` if it is a multiple of the period.

The specification for each functional module includes a clock and a list of input, output and internal variables. All these are time dependent and are represented as functions of the module's clock. For example, an output WSCMD of a module of clock H is of type

```
WSCMD : [H -> ws_range].
```

The elements of H represent the instant when the module is activated, the input signals processed, and the output produced. The clock forms an increasing sequence of instant

$$t_0 < t_1 < t_2 < \ldots < t_i < \ldots$$

and $\mathtt{WSCMD}(t_i)$ is the command computed at the i-th activation.

With this approach, requirements can be specified as a list of function definitions. For example, we could have the following definition for WSCMD:

```
t: VAR H

WSCMD(t): ws_range = max(X(t), Y(t)),
```

where X and Y are other variables of the same module. Since we want the specifications to be implementable in software, certain restrictions are imposed on the form of the definitions. Roughly, we require that the value of an internal or output variable X at time t_i only depends on values of other variables at t_i or at t_{i-1} and possibly on $X(t_{i-1})$. Intuitively, in case where $X(t_i)$ depends on $X(t_{i-1})$, the variable X is kept in memory for use in the subsequent step; the corresponding PVS definition uses recursion. For example a definition such as

```
F(t): RECURSIVE real =
   IF init(t) THEN A(t)
   ELSE F(pre(t)) + A(t)
   ENDIF
 MEASURE rank
```

means that F accumulates the successive values of A. The predicate `init` and the functions `pre` and `rank` are generic and defined for every clock. $\mathtt{init}(t)$ is true if t is the first element of the clock, $\mathtt{pre}(t_i)$ is equal to t_{i-1} and $\mathtt{rank}(t_i)$ is equal to i. The measure clause in the definition is required by PVS for every recursive function and is used to ensure that the function is well defined so that the recursion always terminates.

In most points, this model is similar to an abstract state machine, as used in [3, 5] for example. The state at time t_i is a vector $X_1(t_i), \ldots, X_n(t_i)$ where $X_1, \ldots, X_n$ are variables of the module. The transition function and the initial state are implicitly contained in the definition of all these variables. The restricted form of recursive definition shown above ensures that the state at time t_i depends on the value of input variables at t_i and on the state at t_{i-1}.

Using this form of data flow specifications, the functional requirements are all expressed in a purely definitional style. This is strong evidence concerning the consistency of requirements. When type checking the specifications, PVS may generate various proof obligations known as TCCs. For example, a PVS specification such as

```
cmd_range : TYPE =
  {x: real| 0 <= x AND x < M }

CMD(t): cmd_range = A(t) * A(t),
```

defines a data flow CMD of range `cmd_range`. When analyzing this specification, PVS will generate a TCC to ensure that the expression `A(t) * A(t)` is effectively within the allowed range:

```
CMD_TCC1: OBLIGATION
  FORALL t:
    0 <= A(t) * A(t) AND A(t) * A(t) < M.
```

Provided all such TCCs are discharged, we know that all the data flows are well defined functions; there is no risk of inconsistency. Since PVS requires that all functions are total, this style of specification also ensures the total coverage of input domains.

3.2 Assumptions and Critical Properties

Assumptions about the system under control and the critical properties to be verified are written as PVS axioms and conjectures, respectively. Unlike the functional requirements, assumptions and properties may be related to non-discrete quantities such as external physical parameters or the position of mechanical components. For example, assuming a system receives input signals Pt and Ps from a probe measuring atmospheric pressure then the two inputs might be declared as

```
Pt, Ps: [time -> pressure_range]
```

and we can write assumptions such as

```
pressure_constraint: AXIOM
   FORALL (t:time): Ps(t) <= Pt(t).
```

In the same way, a critical property is written as a conjecture using explicit time indices and quantifiers. For example, a typical response property could look like:

```
response: CONJECTURE
   FORALL t : P1(t) IMPLIES
     EXISTS u :
        t <= u AND u <= t + D AND P2(t);
```

P2 is expected to hold within a delay D after P1 is true.

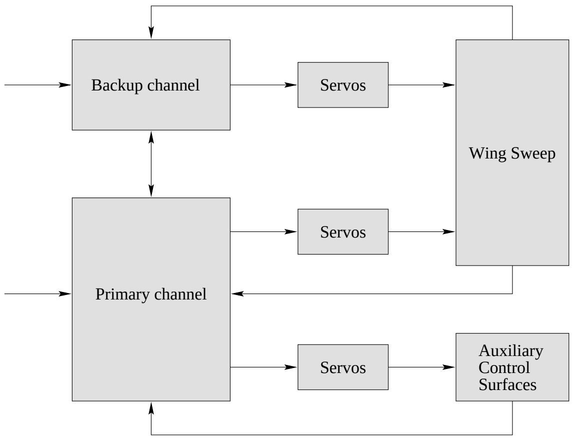

Figure 1: ADC architecture

3.3 Verification

Once formalized, the requirements for a real-time controller consist of one or several functional modules specified as a set of data-flow definitions, a collection of axioms which describe assumptions about the controller's environment, and a list of conjectures which specify critical properties. There are also other elements such as the communication between the modules or between functional modules and the controlled system.

The verification of requirements is then a theorem proving exercise; we have to prove the conjectures using the axioms and the data flow definitions. In practice, the proofs of critical properties can be complex and require the introduction of many intermediate theorems and lemmas.

It is also useful to prove simple properties of the specifications (so-called *putative theorems*). Such properties are intended to check that the formalization of assumptions or functional requirements is reasonable. They are typically simple properties one expects to be true of the system which may or may not help prove the main results.

4 The Case Study

This methodology and formalization approach have been applied within SafeFM to a realistic case study. The system is based on an air-data computer which controls the flaps and variable geometry wings of an aircraft. The example is typical of the target applications. It is a real-time system with a redundant architecture and it performs complex control functions as well as failure detection tasks. Originally, the requirements were expressed in a mixture of English description, mathematical formulas and various graphs. The requirements were produced by software engineers with previous experience with systems similar to the case study and were inspired from a large subset of an existing air-data computer.

The remainder of this section gives an overview of the air-data computer and of its formalization and verification. A more detailed description of the case study appears in [7].

4.1 Architecture

The architecture of the controller is shown in Fig. 1. The system includes two channels with different clocks, each channel being composed of up to four functional modules. A primary channel performs all the controller's function during normal operation and a backup channel with restricted functionality takes over when the primary fails. The two channels do not synchronize; they have two independent clocks of different frequencies. The only cross channel communication is a discrete signal which indicates failures of the primary channel to the backup channel.

The commands from the two channels are transmitted to servos which control the wing sweep actuators. Feedback signals from the servos and actuators are continuously monitored by the two channels in order to detect mechanical failures. The controller receives other input signals from the pilot, from various sensors and probes, and from other avionics systems in the plane.

```
wing_sweep_primary[
     (IMPORTING time, types)
     ALTITUDE : [time-> altitude_range],
     MACH     : [time-> mach_range],
     ...] : THEORY

  BEGIN
  ...

  t : VAR clock[PRIMARY_PERIOD]

  ...

  AUTO_MODE(t) : RECURSIVE bool =
      if     DESELECT_AUTO(t)
      then   FALSE
      elsif  SELECT_AUTO(t) or init(t)
      then   TRUE
      else   AUTO_MODE(pre(t))
      endif
    MEASURE rank
  ...

  WSCMD1(t) : RECURSIVE ws_range = ...

  END wing_sweep_primary.
```

Figure 2: Example of functional module

4.2 Functional Requirements

The PVS formalization of the functional requirements consists of six modules describing the six main functions of the controller. Each of these functional module is specified as a separate PVS theory. Fig. 2 gives an overview of such a functional module. The theory specifies the wing sweep control function performed by the primary channel. The theory parameters `ALTITUDE`, `MACH`, etc. specify the input signals received by the module. The theory contains a list of data-flow definitions which are all functions of the same clock, the clock of the primary channel. The figure shows two examples of data-flows; `AUTO_MODE` corresponds to an internal boolean variable and `WSCMD1` specifies the wing sweep commands produced by the primary channel.

4.3 Assumptions and Safety Properties

The assumptions give a (crude) model of mechanical and electro-mechanical components of the controlled system (sensors, actuators, and interlocks). For example, the axiom `cmd_wings` below relates the wing sweep angle (`WSPOS`) and the wing sweep commands (`CMD`) when certain mechanical interlocks are not active. The value `CMD(t)` depends on the wing sweep command produced by the active channel.

```
cmd_wings : AXIOM
    constant_in_interval(CMD,t,t+eps)
 and
    not wings_locked_in_interval(t,t+eps)
 implies
    CMD(t) = WSPOS(t+eps)
  or
    CMD(t) < WSPOS(t+eps) and
    WSPOS(t+eps) <= WSPOS(t) - eps*min_rate
  or
    CMD(t) > WSPOS(t+eps) and
    WSPOS(t+eps) >= WSPOS(t) + eps*min_rate
```

Other axioms describe the evolution of the wing sweep angle when the locks are active, the initial position of the wings, and various contraints about other components of the system.

The air-data computer has to satisfy two main safety properties related to constraints on the position of the wings when various sets of flaps are extended. Our first attempt to verify these properties failed because of a lack of information about the system under control. In particular, our first model did not include mechanical interlocks which ensure that the flaps do not extend at the wrong time. Other mechanical locks limit the extension of the wings when the flaps are not retracted.

After several iterations and consultation with GEC-Marconi engineers, we reformulated the initial safety requirements in order to take the mechanical locks into account. We defined the *safe states* of the two channels as those where the wing sweep commands issued cannot force the wings against the mechanical interlocks. We then specified three safety properties:

- While the primary channel is in control, the system stays in a safe state.
- After the backup channel assumes control, it converges to a safe state within a specified period of time.
- If the backup channel is in control and in a safe state, it will stay in a safe state.

These three constraints are expressed easily in PVS. The first property ensures that the system is in a safe state until a possible failure of the primary channel. The second corresponds to a transitory period from the instant the primary channel fails to the instant the backup channel reaches a safe state. The third property ensures that once the backup channel has reached a safe state, the system remains in a safe state.

The second property indicates that the backup channel may not be immediately in a safe state when the primary channel fails. This is due to the absence of synchronization between the two channels. Because the two subsystems have different clocks and receive input from separate sources, there can be a substantial difference between their internal states and between the wing sweep commands they compute. If the backup channel takes control while the difference is large, it may momentarily produce commands which force the interlocks but the latter ensure that the wings remain in a safe position.

4.4 Results

The formalization of the system consisted of approximately 4500 lines of PVS specifications. This included basic theories and definitions for time and clocks, a collection of general purpose functions and theorems (*background knowledge*), and the specification of the case study proper. The amount of effort involved in formalization and verification is estimated at around 18 person months. Approximately half of this time was spent on the verification of the three main properties. As a whole, a total of 385 proofs were performed. These include 106 TCCs most of which were discharged automatically by the prover; the rest was proved by hand. The verification of the three top-level critical properties required the proof of a total of 124 propositions. The other properties we verified were putative theorems whose proofs did not pose any major difficulty.

During the proof of a putative theorem, an error was found in our formalization of the functional requirements. This corresponded to an unexpected situation where a "lower limit" gets larger than an "upper limit" and the resulting commands are wrong. This error is triggered by an exceptional combination of input parameters. It can be traced back to the original informal requirements where the possible inversion of the two limits is completely overlooked. This was the only error discovered in the functional requirements. After a simple correction, all the putative theorems were proved.

The main difficulty we encountered was a lack of information about the system under control and the imprecision of the informal safety requirements. There was only limited information about these aspects in the original requirements which were destined primarily for software engineers and focused on functionality. As a result, our first attempts to prove the safety properties were unsuccessful. The solution was to consult with systems engineers who have a wider view of the system including hardware and software as well as safety mechanisms. We also got more information about the expected behavior of the controller from various documents including the pilot's manual. The interaction with systems engineers and the new sources of information helped us clarify the safety requirements and formulate the correct assumptions. Once the precise requirements were established, we were able to prove the safety properties.

The experiment clearly demonstrated the advantage of the PVS specifications over the original informal document; the formal requirements are concise, precise, and unambiguous. In this respect, the SafeFM case study has confirmed other authors' conclusions about the value of formal requirements specifications [3, 5].

However, the main benefits of the formal approach were realised during the validation stages. Proofs are an essential means of detecting errors in requirements. The proof process requires a thorough analysis and gives a profound understanding of the system behavior. In particular, proofs can help understand the subtle interactions between the components of a complex control system.

An important conclusion of the case study is that formal methods can be applied in practice to the requirement analysis of real industrial systems. By the size of its specification and the number of proofs performed, our experiment represents a major effort in formal verification. Such large scale verification is feasible and beneficial provided adequate tool support is available. In this respect, SafeFM has corroborated other reports on industrial uses on PVS, such as [12].

However, another lesson of the experiment was that the proof process can be expensive and difficult to estimate. It took us eighteen months to complete the work instead of the six we had originally planned. Some of these delays can be attributed to our lack of familiarity with PVS. Other factors can also explain this time overrun:

- There was little guidance on how to apply PVS effectively to real-time control applications. A non-negligible part of the work was taken up in writing PVS theories defining the basic notions of clocks and time.

- We had to spent time in developing and proving various general purpose results. For example, we had to prove properties of sets of real numbers or of finite sets.

- The initial requirements document was written from a software engineering point of view and contained little information about the system under control. Several iterations were required before formulating the correct assumptions and getting the precise safety requirements.

5 Building PVS Libraries

Some of the difficulties encountered were due to a lack of maturity of the PVS prover. In order to reduce the effort involved in formal specification and in verification, it is necessary to provide general purpose PVS libraries. Fortunately, this need has been identified and the current version of PVS comes with more predefined notions and with better support for libraries[3]. An important issue for the PVS community is to develop and make available further libraries.

In order to contribute to this effort, we have developed various PVS libraries during the last months of the SafeFM project. The main one includes basic results of real analysis [6]. Some aspects of the case study specification could have been simpler to model if such a library had been available. Other PVS developments were related to finite sets and have been incorporated in a more general library [2]. In more recent work, we defined a smaller set of PVS theories for manipulating roots and square roots of real numbers[4].

The main objectives when building such libraries are generality and usability. We need to define notions as generally as possible to make them applicable to many classes of problems. We also need to present the results in a convenient form to ease their use in PVS proofs.

In developing the SafeFM libraries, we used the following principles:

- For the notion of continuity to be useful in practice, we cannot restrict ourselves to the case of total functions from $\mathbb{R}$ to $\mathbb{R}$; we must be able to consider continuous functions defined on subtypes of the reals.

- The most convenient and powerful way of using lemmas in proofs is using the rewriting capabilities of PVS. The mechanism takes lemmas which have the syntactic form of a "rewrite rule" or of a "conditional rewrite rule". The user can either apply these rules selectively or install them as automatic rules which will be used internally by the theorem prover.

  Another important capability is the general proof commands PVS provides for reasoning by induction. These commands use (explicitly or implicitly) lemmas which must have a particular form. We need to provide such induction rules when necessary.

- PVS relies on a powerful type system and type checking mechanism. The latter can generate proof obligations to ensure that specifications are consistent. As far as possible we need to reduce the number of proof obligations that might be generated when using lemmas and functions from the library. A new feature of PVS – *the judgements* – can be of great help in this respect.

- Taking advantage of the decision procedures is another way of making lemmas and theorems simpler to use; we need to provide results in a form such that the decision procedures can be applied.

  This can sometimes work against the principle of generality. For example, the following important property is proved in the continuity library: from any sequence of reals, one can extract a monotonic subsequence. This property is also true for sequences of any type T, provided T is totally ordered. Unfortunately, manipulations of order relations on general, non-numerical types are extremely tedious because no decision procedures apply.

There are simple pragmatic rules one can adopt which can make libraries both easier to use and widely applicable:

- Generality can be achieved by using parameterized theories and the subtyping structure. For example, parameters allowed us to define continuity of functions of type `[T -> real]` where `T` is any subtype of the reals.

[3] Most of the SafeFM work was done with an older version which did not include these facilities.

[4] These PVS libraries are accessible at
http://www.cs.rhbnc.ac.uk/research/formal/safefm-pvs.html
or ftp://ftp.cs.rhbnc.ac.uk/pub/safefm/pvs/roots.dmp.gz

- Usability can be improved by writing lemmas and properties in certain syntactic forms in order to make them usable as rewrite or induction rules. The possible forms of rewrite rules are presented in [16]; a few examples are given below

```
sqrt_def : LEMMA
   sqrt(x) = y IFF y * y = x

square_sqrt : LEMMA
   sqrt(x) * sqrt(x) = x

sqrt_square : LEMMA
   sqrt(x * x) = x.
```

  The two variables x and y are non-negative reals.

- It is convenient to offer several variants of the same theorem, even if they look redundant. For example, it does no harm to have the following lemmas about square roots even though anything provable with the last three lemmas is provable from the first one alone.

```
both_sides_sqrt_lt : LEMMA
   sqrt(x) < sqrt(y) IFF x < y

both_sides_sqrt_le : LEMMA
   sqrt(x) <= sqrt(y) IFF x <= y

both_sides_sqrt_gt : LEMMA
   sqrt(x) > sqrt(y) IFF x > y

both_sides_sqrt_ge : LEMMA
   sqrt(x) >= sqrt(y) IFF x >= y.
```

  With these four variants, many properties can be proved by automatic rewriting. A single lemma would require much more tedious manual intervention to instantiate the variables.

- The *judgement* clauses are extremely useful for simplifying type checking and reducing the number of proof obligations. This mechanism is described in [13, 4]; it allows us to specify for example that the square root of a positive number is positive:

```
JUDGEMENT sqrt
   HAS_TYPE [posreal -> posreal].
```

- It may be sometimes necessary to reduce generality in order to improve usability. This is in particular the case when notions are defined on types too general for the decision procedures to apply.

All the above rules provide only partial answers to the difficult problem of building usable libraries. It is hard to anticipate what results and lemmas can be useful in practice and to know how to present them in a convenient form. Such issues can only be resolved by acquiring more experience in library development and by learning from users of libraries.

6 Conclusion

One main objective of SafeFM was to investigate practical methods of producing coherent software requirements for digital controllers, using a formal approach and theorem proving. The case study experiment has confirmed the potential benefits of formal requirements analysis. A formal specification provides clear advantages such as clarity and precision, but the main benefit in our experiment was the feasibility of thorough analysis via proofs.

The project showed that PVS provides adequate tool support for such an analysis. A relatively large and complex system was completely formalized and verified with PVS. We believe that the theorem proving technology is mature enough to be applied to real-life, industrial applications. The main practical issue is to provide guidance on the use of theorem provers to specific classes of problems and in developing libraries to facilitate specification and verification.

Acknowledgements

The work described in this paper would not have been possible without the support of the project partners. We are indebted to Tim Boyce, Jonathan Draper, Bob Smith and our other GEC-Marconi colleagues who have had to educate us in avionics systems.

References

[1] J. R. Abrial. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[2] R. W. Butler. The NASA Langley PVS Library, March 1996. This note and the library it describes can be accessed via http://atb-www.larc.nasa.gov/ftp/larc.

[3] J. Crow and B. Di Vito. Formalizing Space Shuttle Software Requirements. In *ACM SIGSOFT Workshop on Formal Methods in Software Practice*, January 1996.

[4] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. In *WIFT'95 Workshop on Industrial-Strength Formal Specification Techniques*, April 1995.

[5] B. Di Vito. Formalizing New Navigation Requirements for NASA's Space Shuttle. In *FME'96*, March 1996.

[6] B. Dutertre. Elements of Mathematicl Analysis in PVS. In *Theorem Proving in Higher Order Logics: 9th Intermational Conference, TPHOLs'96*, pages 141–156. Springer-Verlag, LNCS 1125, August 1996.

[7] B. Dutertre and V. Stavridou. Formal requirements analysis of an avionics control system. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[8] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Data Flow Programming Language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1321, September 1991.

[9] M. S. Jaffe, N. G. Leveson, M. P. E. Heimdahl, and B. E. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Trans. on Software Engineering*, 17(3):241–258, March 1991.

[10] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice-Hall International, 1986. 2nd Edition.

[11] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming Real-Time Applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[12] S. P. Miller and M. Srivas. Formal Verification of the AAMP5 Microprocessor: A Case Study in the Industrial Use of Formal Methods. In *WIFT'95 Workshop on Industrial-Strength Formal Specification Techniques*, April 1995.

[13] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[14] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language*. Computer Science Lab., SRI International, April 1993.

[15] D. L. Parnas. Some theorems we should prove. In *Proc. of 1993 International Meeting on Higher Order Logic Theorem Proving and Its Applications*, pages 156–163. The University of British Columbia, Vancouver, BC, August 1993.

[16] N. Shankar, S. Owre, and J. M. Rushby. *The PVS Proof Checker: A reference Manual*. Computer Science Lab., SRI International, March 1993.

[17] V. Stavridou, A. Boothroyd, T. Boyce, P. Bradley, J. Draper, B. Dutertre, and R. Smith. Developing and Assessing Safety Critical Systems with Formal Methods: the SafeFM Way. *Journal of High Integrity Systems*, 1(6):541–545, 1996.