Formal Requirements Analysis of an Avionics Control System

To Appear in IEEE Transactions on Software Engineering, Vol. 23, No. 5, 1997

Bruno Dutertre, Victoria Stavridou

Abstract—**We report on a formal requirements analysis experiment involving an avionics control system. We describe a method for specifying and verifying real-time systems with PVS. The experiment involves the formalization of the functional and safety requirements of the avionics system as well as its multilevel verification. First level verification demonstrates the consistency of the specifications whilst the second level shows that certain system safety properties are satisfied by the specification. We critically analyze methodological issues of large scale verification and propose some practical ways of structuring verification activities for optimising the benefits.**

Keywords—**Formal specification, formal verification, safety critical systems, requirements analysis, avionics systems.**

I. Introduction

THIS paper reports on an experiment in the use of formal methods for producing and analyzing software requirements for a safety-related system. This work was conducted as part of the SafeFM project [3], [4], a collaboration with GEC Marconi Avionics (Mission Avionics Division) and AEA Technology (Consultancy Services). The SafeFM project was intended to support the practical use of formal methods for high integrity systems not by producing new theories but by integrating formal methods into existing development and assessment practice. The project focused on a particular class of application – real-time control systems – and most of the work is based on an avionics case study; a digital system controlling the variable geometry surfaces of an aircraft.

This paper describes the application of a formal approach to the specification and analysis of the SafeFM case study requirements. The case study is a realistic system inspired by an existing air data computer (ADC). It is a real-time control system which consists of two independent control channels: A primary channel performs all ADC functions during normal operation and a backup channel takes over when the primary fails. The two channels perform complex control and failure detection functions and have to satisfy safety-critical properties.

The work on the case study was supported by the PVS specification and verification system [5], [6]. We specified the functional requirements of the case study in PVS using a data flow approach. The purely definitional style adopted and the strong typing mechanisms of PVS give us strong assurance concerning the internal consistency of the functional specifications. In addition we performed various verifications by proving so-called *putative theorems*. Type-checking and putative theorems correspond to a first level of validation. On a second level, we verified that the safety-critical requirements were satisfied. This requires a system-wide perspective; relevant aspects and properties of the system under control and of the physical environment have to be included. For this purpose, we used an approach based on explicit time which is easy to implement in PVS. All the assumptions and safety properties were written in PVS and the verifications were performed interactively with the PVS theorem prover.

The potential of formal methods to improve system dependability has been recognized by several certification authorities; their use is recommended or mandated by emerging standard in the safety critical sector [1], [2]. The main objective of this work was to get an indication about the feasibility, benefits, and limitations of a formal approach when applied to a realistic example of substantial size and complexity.

The remainder of this paper is structured as follows: Section II gives an overview of the SafeFM case study. The main functional and non-functional requirements are summarized together with the architecture and fault-tolerance features of the system. Section III presents PVS and the main formalization principles adopted for this experiment. Section IV describes the successive stages of the case study, that is formalization of the functional requirements, formulation of safety properties and assumptions, and verification. In section V we draw lessons from the experiment. We discuss the perceived benefits and limitations of our formal approach and the adequacy of tools for the experiment. We identify several sources of difficulty and inefficiency and we propose possible improvements.

II. The case study

The SafeFM case study is an avionics controller inspired by an existing air data computer (ADC) embedded in a fighter aircraft. The system computes air data parameters such as altitude, airspeed, or Mach number and is also in charge of several aerodynamical control surfaces of the plane. The case study concentrates on one of the most critical functions of the ADC, the control of the variable geometry wings. The wings can be swept aft or forward in order to optimize flight performance. The ADC computes an optimal wing sweep angle according to values of aero-

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

The authors are with the Department of Computer Science, Queen Mary and Westfield College, University of London, Mile End Rd, London E1 4NS, UK. E-mail: bruno@dcs.qmw.ac.uk, victoria@dcs.qmw.ac.uk

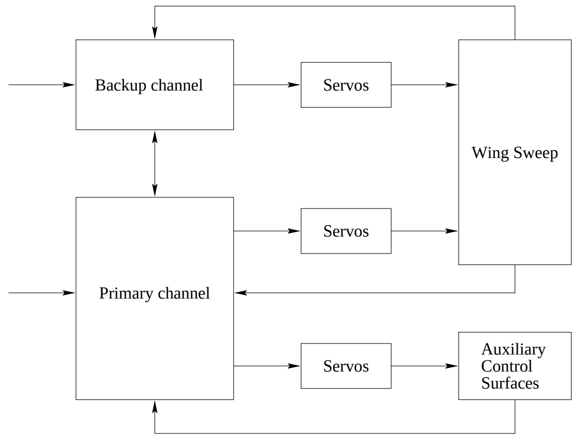

Fig. 1. ADC architecture

dynamical parameters, commands from the pilot as well as mechanical constraints.

In order to tolerate mechanical failures, the wing sweep actuators can be controlled by two independent servos. The ADC must detect and signal a possible failure of the servos and is required to tolerate such a failure. Accordingly, the ADC is composed of two control channels, each connected to a different servo. The architecture which is shown in Fig. 1 is asymmetric; a primary channel performs all the ADC functions during normal operation and a backup channel takes over when the primary fails. When both channels fail, the pilot can still control the wing sweep using an emergency mechanical device. The two channels are independent; they have separate power supplies and receive input from different sources. There is no synchronization between the two channels which have two independent clocks of different frequencies. The only cross channel communication is a discrete signal indicating failures of the primary channel to the backup channel. Wing sweep commands are continuously computed by the two channels but the servos can be individually enabled or disabled to ensure that only one channel is in control at a time. The ADC must ensure that, in normal mode, only the primary servo is enabled and that the backup servo is only enabled when the primary channel fails.

The two channels detect failures by continuously monitoring feedback signals from the servos and actuators. When the difference between the wing sweep commands and the responses from the servos and actuators exceeds a threshold for a long enough period, the mechanical elements are deemed to have failed. The ADC then reports the failure to the pilot and the faulty channel is deactivated. A faulty channel continues normal computations but the associated servo stays disabled. The only way to reactivate a channel is for the pilot to reset the whole system. This clears the failure monitoring mechanisms and restores the initial configuration of the ADC (the primary channel active, the backup servo disabled).

The main safety requirement of the system is to prevent contact between different sets of flaps located on the trailing edge of the wings and the fuselage which can severely damage the flaps and lead to loss of mission. In order to prevent this, the ADC maintains the wing sweep angle below certain limits when the flaps are extended. For this purpose, the ADC receives signals from mechanical sensors which indicate whether the flaps are extended or not.

The case study is a realistic example of the class of applications the SafeFM project aimed to address. It is inspired by an existing system already developed by GEC-Marconi which has been in service for several years. The case study is typical of avionics control systems. It is a real-time system with a redundant, fault tolerant architecture; it communicates and interacts with a complex environment; and it has to meet safety and mission critical requirements. All of these facts, as well as the dual channel architecture, render the ADC substantially complex.

The software requirements of the case study describe the various control and failure detection functions implemented by the two channels. The wing sweep commands are computed according to complex control laws combining numerical calculations and logical constraints. Specific commands depend on many parameters such as the altitude and speed of the plane, the position of the flaps and other mechanical elements, and the control mode selected by the pilot. On the other hand, the timing requirements are relatively simple. They correspond to delays in the failure detection mechanism.

Originally, the software requirements were expressed as a mixture of English descriptions, mathematical formulas and various graphs [7]. Our objective was to produce more precise and rigorous specifications of these requirements and to adopt a multilevel verification approach for increasing our confidence in their correctness. First, we wanted to check the internal consistency of the specifications to ensure that the various requirements did not contradict each other. The second and most important level of verification involved checking that the safety critical properties were satisfied, that is, the ADC effectively prevents contact between the flaps and the fuselage.

III. Methods and tools

A. Needs

Analyzing a control application such as the ADC requires considering both the controller and its environment. The safety requirements are properties of the wings and flaps and cannot be verified without knowledge of the system under control. We have to make assumptions about the wing sweep sensors, servos, and actuators, as well as about the behavior of the different flaps. Formalizing these assumptions requires a notation capable of representing continuous evolution of physical and mechanical variables. To model the controller itself, we have to specify the software requirements, that is, the various control and failure detection functions to be implemented. We also need to consider other elements such as communication between the two channels or the presence of two different clocks in the system.

In general, specifications for applications such as the ADC can be structured into the controller specifications, a model of the system under control, and the critical properties we want to verify. We need an expressive specification language able to cover all three aspects. The applications we are interested in are real-time, interactive systems. They are also hybrid systems since they involve both physical variables and digital components. The formalism must allow us to manipulate both continuous and discrete variables, to express quantitative timing requirements, and to deal with multiple clocks.

Verification of applications of the size and complexity of the ADC are only practical with tool support. An efficient mechanical proof tool is needed in order to make verification possible. We chose to use PVS because of its powerful theorem prover and of its rich specification language.

B. An introduction to PVS

PVS is a specification and verification environment. It integrates tools for the creation and analysis of formal specifications and an interactive theorem prover [6], [8]. PVS has been used successfully to produce substantial and complex proofs of hardware and fault tolerant clock synchronization algorithms [9], [10], [5]. It has also been used in other domains such as compiler verification [11] or the construction and validation of real-time systems [12], [13].

PVS specifications are written in an extension of classical higher-order logic. They are organized in a hierarchy of theories which can be parameterized. A theory usually contains type, constant, or function definitions, and a collection of lemmas and theorems. It is also possible to state assumptions about theory parameters and to introduce axioms. Once PVS has analyzed the specifications for syntax and type correctness, the user can attempt to prove theorems. The system provides a LaTeX generator and various other tools for the analysis of specifications and proofs.

PVS has a rich type system which includes constructors such as tuples or records and supports dependent types, abstract data types, and a powerful subtyping mechanism. These features make the language very expressive and facilitate specification but type checking becomes undecidable. Proof obligations known as Type Checking Conditions (TCCs) may be generated by the type checker and have to be discharged in order to establish type consistency. In most cases, the proofs of TCCs can be performed automatically by the PVS theorem prover, but in more complex situations, manual assistance may be required.

The PVS theorem prover is based on the sequent calculus. Starting from an initial sequent, the user progressively develops a proof tree by applying commands and deductive rules. At every stage in the proof, a particular sequent – the *goal* – is displayed and all proof commands apply to this sequent. Commands can either complete the proof of the current goal or expand the tree by generating subgoals which in turn have to be proved. Proof rules are available for propositional and quantifier reasoning and PVS provides a set of high level commands for more complex reasoning. For example, there are commands for induction, for manipulating automatic rewrite rules and for the heuristic instantiation of quantifiers. The system also includes powerful simplification and decision procedures for linear arithmetic and equality. All the commands can be combined to form proof strategies, allowing several proof rules to be applied in one step.

PVS specifications are built from a collection of standard types, functions, and constants defined in the *prelude*, a set of primitive theories. The prelude also introduces fundamental axioms and provides a set of pre-proved theorems. The standard numerical types, that is, reals, rationals, integers and natural numbers, are all primitive and are defined in the prelude. They are organized in a hierarchy; the natural numbers form a subtype of the integers, which are a subtype of the rationals, which are a subtype of the reals. Unlike other systems such as HOL [14], PVS does not construct the numerical types from below, starting with the natural numbers but defines their properties axiomatically. Numbers and other important types such as sets, lists, and sequences are pre-defined in the prelude.

C. Real-time specifications in PVS

Several authors have proposed different ways of applying PVS to real-time systems. A method based on an extension of Hoare triples is presented in [12]. The approach allows the construction of correct real-time systems by successive refinements. PVS is used to define the semantics of the formalism and to implement proof and refinement rules. Another application of PVS to the specification and verification of real-time programs is described in [13]. Program behaviors are represented by infinite sequences of states which include quantitative timing information; each state is labeled by a time of occurrence. Temporal operators defined in PVS are used to assert properties of these sequences.

PVS also provides tool support for the duration calculus (DC), a real-time interval temporal logic [15], [16]. DC has been partly embedded in PVS. Several theories define the semantics of the calculus and DC proof rules are introduced as PVS theorems. A dedicated interface makes the encoding largely transparent to the user. Specifications are written in a syntax close to standard DC and verification is performed using the DC proof rules directly [17].

The first two methods are mostly designed for program verification and construction. They assume that systems evolve in a step by step fashion and can be characterized by a succession of states. This discrete model does not suit our purpose very well. The duration calculus and its extensions (see [18], [19]) are intended for the development of embedded systems and use continuous time. However the DC tends to focus on control-intensive applications with complex timing requirements. The logic includes modalities which can express temporal properties without explicit usage of time variables. The case study does not require such a sophisticated timing formalism. Instead our specifications are based on a straightforward and easy to implement approach; time-dependent quantities are manipulated explicitly as functions of time and temporal properties are written using explicit time indices.

The model and terminology we use are inspired partly from the semantics of the duration calculus, partly from the real-time data flow languages LUSTRE and SIGNAL [20], [21]. We distinguish between two classes of time-dependent variables. Some are similar to the state variables of DC, others are closer to the notion of data flow of LUSTRE and SIGNAL.

The state variables are total functions from the time domain to a value domain. They represent the continuous variables of the application, that is, the physical parameters such as altitude, the mechanical variables such as wing sweep angle and the input signals received from the pilot.

The data flows are used to formalize the software requirements and model the discrete behavior of the digital components. We assume that these components have a fixed interface composed of a finite number of input and output ports and that they are activated at regular intervals by a clock of known frequency. During each activation phase, a component performs internal computations and delivers a new value on all its output ports. The behavior of a digital component is then characterized by the sequences of values produced on its output ports and by its clock which determines when the values are produced. We represent clocks by infinite sets of instants; every element of a clock can be interpreted as an instant of activation of the component. Output signals as well as internal variables used in the specifications are modeled by data flows, that is, functions from the clock to some domain of values. For example, the wing sweep commands computed by the primary channel form a data flow `WSCMD1` defined at successive instants

$$t_1 < t_2 < \ldots < t_n < \ldots$$

of the primary clock, and `WSCMD1`(t_n) is the command produced at the n-th activation of the primary channel.

The last elements needed in the specification are conversion functions which transform discrete data flows to state variables. Such conversions are used to model communication between the discrete components and the physical environment. For example, the wing sweep angle is a physical variable represented by a total function of time `WSPOS` and we have to describe how `WSPOS` relates to the discrete commands from the two channels. For this purpose, we convert the data flow `WSCMD1` above and a similar `WSCMD2` from the backup channel to piecewise constant state variables `CMD1` and `CMD2`. Depending on the status of the two channels, the actual command `CMD(t)` transmitted to the wings is either equal to `CMD1(t)` or to `CMD2(t)` and we model the actuator by assumptions relating `CMD` and `WSPOS`. In effect, the conversion functions give us a convenient way to mix data flows of different clocks such as `WSCMD1` and `WSCMD2` into a single state variable `CMD`. Conversions can also have a similar use for modelling communication between software modules with different clocks.

The case study specifications are based on a set of PVS theories which introduce the notions of time and clocks. The definitions rely on PVS subtyping facilities; the time domain is a subtype of the reals and a clock is a parameterized subtype of the time domain. The basic theories also introduce operators to facilitate the definition of data flows and several predicates to simplify the expression and verification of properties. In addition, we also needed a few general purpose theories which provide background knowledge. For example, these theories contain properties of the real numbers or theorems about bounded or finite sets and define functions such as the absolute value or the maximum or minimum of two reals.

C.1 Time domain

As in the duration calculus, we use the non-negative reals as time domain with 0 as the initial instant. The corresponding type, `time`, is defined as follows:

```
non_neg : TYPE = { x : real | x >= 0 } CONTAINING 0

time : TYPE = non_neg.
```

The first line introduces the non-negative reals as a subtype of the pre-defined type `real` and `time` is declared synonymous to `non_neg`. The containing clause specifies that `non_neg` is a non-empty type. This simplifies type checking and reduces the number of TCCs generated by PVS.

C.2 State variables

The state variables are declared and manipulated as PVS functions. For example, a state variable representing the position of the wings is declared as follows:

```
WSPOS : [ time -> ws_range ],
```

that is, as a function from `time` to `ws_range`. The latter type represents the domain of values for the wing sweep angle and is defined as a subtype of the reals.

Assumptions about the system under control are introduced as axioms in the specifications. They correspond to constraints or relations between the values of state variables. For example, there is an upper bound on the rate of rotation that the wing sweep actuator can produce. This assumption is formalized by the axiom `variation_ws` below.

```
max_rate : posreal

t : VAR time

delta : VAR posreal

variation_ws : AXIOM
   abs(WSPOS(t + delta) - WSPOS(t)) <= max_rate * delta.
```

The axiom specifies that the constant `max_rate` is the maximal rate of variation of the wing sweep angle. In the axiom, as in any PVS formula, the free variables `t` and `delta` are considered universally quantified.

C.3 Clocks

The specifications rely on a simple model of clocks. All clocks are assumed to have a fixed period which can be any positive number. A clock of period K is represented by the set of successive multiples of that period, starting from 0. In PVS, a clock is a sub-type of `time` defined in a parameterized theory `clocks`:

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

   IMPORTING time

   clock : TYPE =
      { t: time | EXISTS (n : nat) : t = n * K }
```

The parameter `K` is a strictly positive real and represents the clock period. The definition specifies that an instant `t` is of type `clock` if and only if it is a multiple of `K`.

Several basic operations on clock elements are useful for constructing and reasoning about data flows. Like the type `clock`, these functions are parameterized by a clock period and are defined in the theory `clocks`. Among these operations, the following three are fundamental:

- `init` indicates whether a clock element is the initial instant of the clock,
- `pre` gives the immediate predecessor of an element and
- `next` gives the immediate successor of an element.

In PVS, all functions have to be total, so that `pre` is a function defined on the sub-type of `clock` that excludes the initial element. The three functions are defined below:

```
x : VAR clock

init(x) : bool = (x = 0)

noninit_elem : TYPE = { x | not init(x) }

y : VAR noninit_elem

pre(y) : clock = y - K

next(x) : noninit_elem = x + K.
```

When type checking these definitions, PVS generates two proof obligations. The first requires showing that `y - K` is of type `clock` to ensure that the definition of `pre` is consistent. This TCC states that for any `y`, the expression `y - K` is non-negative and is a multiple of `K`. The second TCC is similar and ensures the type correctness of `next`.

The elements of a clock form an ascending sequence of instants and another useful function assigns to any clock element x its rank in this sequence, that is, the unique natural number n such that $x = nK$:

```
rank(x) : nat = x/K.
```

Here again, PVS requires showing that the division yields a natural number.

A number of other functions similar to `pre` and `next` are defined in `clocks`. The theory also contains several lemmas and theorems which correspond to elementary properties of clocks. For example, a lemma asserts that a clock is not bounded and another states that any time t is between two successive clock elements. The most important theorems proved in `clocks` are general induction rules such as the following:

```
clock_induction : PROPOSITION
  FORALL (P : pred[clock]) :
    (FORALL (t : clock) : init(t) IMPLIES P(t))  AND
    (FORALL (t : noninit_elem) : P(pre(t)) IMPLIES P(t))
      IMPLIES (FORALL (t : clock) : P(t)).
```

This simply means that the invariance of a predicate `P` – a function from `clock` to the booleans[1] – can be shown by proving that `P` holds initially and that it is maintained after every clock step. The proposition is easily derived from the pre-defined induction theorem for the natural numbers using the function `rank`.

To summarise, the theory `clocks` includes a set of elementary functions and a list of useful properties of clocks. This is the basis for formalizing the software functional requirements. In particular, the main mechanisms we used for expressing and verifying the functional specifications – recursion and induction – are supported by functions and properties developed in `clocks`.

C.4 Data flows

The software requirements of the case study are structured in modules, each of which is a PVS theory specifying a function of the ADC. For example, a module describes the primary wing sweep control and another describes the backup failure monitoring. Each module has a clock and a list of input signals, and defines a collection of data flows representing output or intermediate variables. Every module is assigned to one of the two channels which determines the clock of the module. The input signals are either state variables the channel receives from the external environment or data flows defined in other modules of the same channel. We use the PVS importing mechanism to specify the clock of a module and the input signals are given as parameters to the theory.

All the data flows defined in a module are functions from the module's clock to a data type. Since each module is expected to be software implementable certain restrictions are imposed on the form of data flow definitions. Roughly, a module with data flows $X_1, \ldots, X_n$ is intended to specify an abstract state machine whose state at clock time t is the vector $X_1(t), \ldots, X_n(t)$. The behavior of such a machine is specified by initial conditions and by a transition relation which defines the state $X_1(t), \ldots, X_n(t)$ as a function of the previous state and the current value of input variables. The definition of $X_i(t)$ is only allowed to refer to the previous or current clock times. In general, if t is not the initial instant of the clock, the value of X_i at time t can depend on

- the current value of input signals,

[1]The type `pred[clock]` is equivalent to `[clock->bool]`.

• the current or previous values of flows X_j where $j \neq i$, or
• the previous value of X_i.

In many cases, data flows are specified by simple PVS definitions such as

```
FWD_LIMIT(t) : wsch_range = min(AFT_LIM(t), FWD_LIM(t))
```

In this example, `wsch_range` is a subtype of the reals and PVS generates a TCC to check that the right-hand side expression is of the allowed type.

In more complex cases where $X_i(t)$ depends on $X_i(\texttt{pre}(t))$, the data flows are defined recursively. In PVS, the definition of a recursive function takes a special form. One has to provide a *measure* used to decide whether the function is well defined, thereby ensuring that the recursion always terminates. The measure assigns to the arguments of the function a value of a well ordered type – usually the natural numbers – and the recursion is sound if it can be shown that the measure is strictly decreasing with every recursive call. The type checker generates corresponding proof obligations called *termination TCCs*.

Recursive data flows are all based on the functions `init` and `pre` associated with the clock and they all use the function `rank` as their measure. The following definition is a typical example of the recursive constructions in the case study:

```
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
```

The boolean flow `AUTO_MODE` represents a state variable controlled by two other flows `SELECT_AUTO` and `DESELECT_AUTO`. By default, `AUTO_MODE` is initially true but this can be overridden by `DESELECT_AUTO`. Subsequently `AUTO_MODE` is set according with the two control flows and it remains unchanged when both are false. The termination TCC associated with this definition is the following:

```
AUTO_MODE_TCC2: OBLIGATION
   (FORALL (t : clock):
       NOT DESELECT_AUTO(t) AND
            NOT (SELECT_AUTO(t) OR init(t))
                 IMPLIES rank(pre(t)) < rank(t));
```

This can be easily discharged and the definition of `AUTO_MODE` is sound. Since all the recursive data flows have the same measure and their definitions are similar, the terminations TCCs can always be discharged using the lemma:

```
clock_recur : PROPOSITION
   not init(x) implies rank(pre(x)) < rank(x)
```

This property is proved once and for all in the theory `clocks`.

Using this form of data flow specifications, the software functional requirements are all expressed in a purely definitional style. This is strong evidence concerning the consistency of the requirements. Provided all the TCCs are discharged, we know that all the data flows are well defined functions and hence there is no risk that parts of the functional requirements are contradictory. Since all the functions are total, there is no risk of incompleteness either; for any combination of values on the input signal, all the internal and output variables have a specified value.

On the other hand, this form of specification can reduce the user's flexibility in writing requirements. Some of the restrictions are due to the rules of the PVS specification language; for example, PVS does not allow the definition of mutually recursive data flows. The other restrictions are due to the underlying state machine model: a data flow definition can only refer to the current or previous instant. These restrictions are important for getting confidence in the feasibility of the software requirements. The state machine model is in fact adopted by many approaches to software development, including recent work with PVS on the specification of the space shuttle software [22]. The state machine model is also widely used by formalisms such as B or VDM [23], [24] which incorporate software development methods.

As a whole, the data flow specification of software requirements provides a solid basis for subsequent development. Type checking ensures completeness and consistency and we feel that the restrictions on the form of specifications are a price worth paying considering the potential benefits.

IV. The experiment

The work on formalisation and verification of the case study was spread over a period of two years. The experiment went through several successive phases, from the first specifications of the functional requirements to the formal verification of the last safety property.

A. Functional requirements

The first stage of the work was the formal specification of the functional software requirements. Our starting point was a report describing informally the ADC architecture and the main functions of the system [7]. This original document describes the two channels and their interfaces and specifies the redundancy management mechanism. The report also specifies in fairly precise terms the different control and failure detection functions to be implemented. The notation used is a mix of English descriptions, mathematical formulas and equations, and various tables and graphs. This document was produced by software engineers with previous experience with systems similar to the case study. The requirements therein were inspired from a large subset of an existing avionics system developed by GEC-Marconi.

The main effort during the early phases of the work focused on defining the specification approach and on developing PVS support theories. The choice of the data flow style was largely motivated by the form of the informal requirements and was made rapidly. However, the theories defining time domain and clocks underwent several modifications before reaching their final form. In particular, we started with a general model of clocks which required a more complex PVS definition than given in section III-C.3; a clock was any unbounded, countable set of instants [25]. This general model did not prove convenient or useful and was later on largely simplified. The basic support theories have been evolving throughout the whole project but a fairly stable version was obtained at the end of the formalization of the software requirements. At that time, the definitions of time domain and clocks, and the clock operators were in the form described in the preceding section. The only modifications introduced in later phases were the adjunction of lemmas needed to perform the safety verification.

Once the building blocks of the formalization had been identified, specification of the functional requirements was straightforward. The informal requirements translated to a list of data flow definitions without any difficulty. This was certainly due to the quality and precision of the original document [7]; we were also aided by a VDM specification of the case study which had been developed separately within the SafeFM project [26]. The VDM specification and direct contact with GEC-Marconi resolved the few ambiguities we found in the informal requirements document.

The structure of the PVS specifications follows closely the informal document organization. Six software modules are described, four for the primary channel and two for the backup channel. Each of these modules corresponds to one main function of the ADC and is specified in a separate PVS theory.

Fig. 2 illustrates the general structure of these functional modules. The main data flow specified in the theory is `WSCMD1`, the wing sweep command produced by the primary channel. This data flow depends on various input signals to the modules, such as `ALTITUDE` or `MACH`, and on intermediate data flows such as `AUTO_MODE`. All the functions defined in `wing_sweep_primary` are discrete flows of clock `clock[PRIMARY_PERIOD]`, the clock of the primary channel.

Type checking the functional specifications was straightforward. Most of the TCCs generated were simple properties which could be discharged automatically by the theorem prover. The others were termination TCCs associated with recursive definitions of data flows.

The simple fact of proving type correctness of the specifications gives us a first level of confidence in their consistency and completeness. It also ensures that simple errors such as out of range values or division by zero are eliminated. However, in order to get greater confidence in the functional specifications, we also need to check high level safety properties.

B. Assumptions and safety properties

The second major step in the SafeFM experiment was to formalize the safety properties and to model the system under control. The two main safety requirements selected for the case study are upper limits on the wing sweep angles when two different sets of flaps are extended. These two properties were given in the original requirement document [7] and were extracted from a Failure Mode and Effect Analysis performed on the system which inspired the case study. The safety requirements are properties of the system under control and cannot be verified without making assumptions about the controlled system. They are expressed in terms of physical variables, namely, the wing sweep angle and the degree of extension of the different flaps. We need to describe how these external variables relate to the input and output signals of the ADC.

```
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.
```

Fig. 2. The general organisation of functional modules

Our first attempts to prove that the safety properties were satisfied were largely unsuccessful. The origin of the problem was lack of information about the components of the system under control, in particular the flaps; anarchic behavior of the flaps makes it impossible for the ADC to ensure the safety requirements. The wing sweep control is implicitly based on the assumption that the flaps do not extend at the wrong time. Unfortunately, there is very little information about the flaps in the requirements document. The original requirements were destined primarily for software engineers and they focus on functionality and on software aspects. They contain only limited information about the system under control because this information is not needed in order to develop the software. We believe that this problem is pervasive in the verification of such systems and a methodology is needed for overcoming it. We cannot prove the safety properties of the software without changing the way overall requirements are traced to system components.

The solution was to consult with the systems engineers who have a much wider view of the system, including hardware and software components as well as safety mechanisms. We also got more information about the expected behavior of the wing sweep and flap control systems 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.

New elements, in particular mechanical interlocks, were included in our model of the controlled system. These locks were not mentioned in the initial requirements, although some of them play an essential role since they prevent the extension of the flaps when the wings are swept over certain limits. This was a crucial assumption that enabled the verification of the main safety properties.

All the assumptions are formalized as axioms in the PVS specifications. For example, the following assumption relates the wing sweep angle `WSPOS` at time `t + eps` and the wing sweep command `CMD` at time `t`, in case none of the interlocks is active.

```
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 * ws_min_rate
  or
     CMD(t) > WSPOS(t + eps) and
     WSPOS(t + eps) >= WSPOS(t) + eps * ws_min_rate
```

Other axioms describe the evolution of the wings when the locks are active, the constraints on the flap extensions, and the initial position of the wings and flaps.

The new assumptions about the system under control posed a different problem. The role of the mechanical interlocks is to prevent the violation of the main safety properties in case the ADC fails. Consequently, the original safety properties derived from the FMEA are trivially satisfied. We faced the following dilemma:

- The interlock assumptions are sufficient to guarantee the safety properties, whatever the behavior of the ADC, but
- without these assumptions, the safety properties are not satisfied.

In order to check the ADC's functional specifications, we had to formulate other safety properties. The solution was to reformulate the safety properties not directly in terms of wings and flap positions but as constraints on the wing sweep commands issued by both channels. The reworked safety requirements state that the ADC must not issue commands that would force the wings against the mechanical locks.

A final difficulty was taking into account the absence of synchronization between the two channels. In certain circumstances, this can cause a substantial difference between the wing sweep commands computed by the backup and the primary channel. If the backup channel takes control while the difference is large, it may momentarily produce commands that will force the locks.

In the end, the safety requirements were refined in three properties:

- While the primary channel is in control, it maintains the system 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.

Safe states are those where the wing sweep commands produced cannot exceed the limits permitted by the mechanical locks. The first property ensures safety from the initialization of the system until a possible failure of the primary channel. The second property corresponds to a transition phase from the instant the primary channel fails to the instant the backup channel reaches a safe state. During this phase, the channel may temporarily issue commands which force the interlocks but the latter ensures that the wing sweep angle does not exceed the limits. The third property ensures that once a safe state is reached, the backup channel will maintain safety.

All these properties are formalized easily in PVS. For example, the first safety property above is written:

```
safety_condition1 : COROLLARY
   CORRECT_PRIMARY(t1) implies
      FORALL (u : primary_time) :
         u <= t1 implies SAFE_PRIMARY(u).
```

C. Verification

The final phase of the case study experiment involved the formal verification of properties of both the software functional specifications and of the whole control system. Two categories of lemmas and theorems were proved. Some were *putative theorems* which allowed us to check the formalization whilst others were directly used in the proof of the three main safety properties.

The putative theorems were either related to the software specifications or to the model of the system under control. Typically, they express simple properties that we expected to be true of the system; thus providing a simple means of ensuring that the formalization was reasonable. They also constituted a collection of lemmas which simplified the verification of the most complex properties. None of these theorems posed any particular difficulty.

The following lemma is typical of these putative theorems. It states that successive commands computed by the primary channel are within a specified rate limit.

```
rate_lemma1 : LEMMA
   FORALL t :
      init(t) or RESET(t)
   or abs(WSCMD1(pre(t)) - WSCMD1(t)) <= RATE_LIMIT(t).
```

The remainder of the verification focused on establishing the three main safety properties. The proofs of these properties were significantly more complex than those of the putative theorems. The three safety theorems required the formulation and proof of several dozens of lemmas and sublemmas. The verification of the three top level safety properties required the proof of a total of 124 propositions.

The least straightforward of the three safety properties was the convergence of the backup channel to a safe state. The proof is based on case analysis; several modes of evolution of the whole system were identified and analyzed separately and then the possible transitions between different modes were examined. Each mode is characterized by parameters, such as the status of the flaps (extended or retracted), the position of the wings, and the internal state of the backup channel (the past values of relevant data flows). Lemmas showed that within each mode, the global system always evolves smoothly towards a safe state. Analysing the transitions showed that this property is preserved as a whole. Although simpler, the proofs of the other two safety properties are based on a similar form of reasoning and are essentially elaborate case analyses.

D. Results

During the proofs, one error was found in our formalization of the functional requirements. The wing sweep command definition for both channels relies on a lower and an upper limit. For certain combinations of the input parameters, the lower limit is greater than the upper limit and this leads to an incorrect wing sweep command. This unexpected situation was discovered by failing to prove a putative theorem. The error can be traced back to the original informal requirements where the possible inversion of the two limits is completely overlooked. The same mistake was present in the VDM specifications given in [26] which were derived from the same informal requirements. This was the only error discovered during the proofs. After a simple correction, the putative theorems and the three important safety properties were all formally proved.

The whole specification consists of approximately 4500 lines of PVS (with comments and blank lines). This includes the library of basic theories for time and clocks, the formal specification of the functional requirements, and a top-level theory which contains the assumptions and all the lemmas and theorems needed to verify the three main safety properties. A total of 385 proofs were performed. These include 106 proof obligations (TCCs) most of which were discharged automatically; the rest was proved by hand.

The amount of effort involved is estimated to around 18 person months. Approximately 6 person months were necessary to write the formal functional specifications and the support libraries. The remaining 12 person months were spent on formalizing the assumptions and carrying out the verification. The bulk of the verification work was spent on the three main safety properties which consumed 9 person months of effort

V. Lessons

A. Benefits of formal requirements analysis

The case study analysis involved two separate activities; the formal specification of the ADC functional requirements and the validation of these specifications through proof. The experiment clearly demonstrated the benefits of the PVS specifications over the original informal document; the formal requirements are more concise, precise, and unambiguous. We are also confident through the use of type checking and because of the definitional style adopted that the PVS specifications are consistent. In this respect, the SafeFM case study has confirmed other authors' conclusions about the value of formal specifications (for example, see [22], [27]). The existence of the VDM specification and the substantial experience of the GEC-Marconi engineers with the system, meant that the PVS formalization was not likely to uncover previously unknown problems.

The main benefits of the formal approach were realised during the later stages of validating the specifications. As the case study illustrated, formal specifications can be clear, unambiguous, and consistent; they can also be wrong. Proofs provide an essential means of detecting errors in the requirements. More generally, the proof process requires a thorough analysis of the requirements and gives a profound understanding of the system behavior. Among the positive results of the proofs attempted on the case study we can cite:

• *A clarification of external conditions for the ADC to work correctly.* The controller requirements are based on implicit assumptions about the system under control and about elements such as interlocks and flaps. Early attempts to check safety properties failed because of lack of information about these external elements. The explicit formulation of the assumptions was an essential step in understanding the control system.
• *A deep analysis of rare and unexpected behaviors of the system.* Among these we can cite the behavior of the ADC after the primary channel has failed. From the initial requirements, it was certainly not clear how the global system would behave in this case. The formal analysis showed that the backup channel may be momentarily in an unsafe state but will eventually reach a safe state. The analysis also gave us a bound on the time this would take.
• *The detection of an error in the formal requirement specifications.* The wing sweep command defined in the PVS specifications was wrong in some circumstances. Proofs are an effective means of detecting such errors at an early stage in the software development. The error was also found in the VDM specification via animation [28] independently of its discovery by verification. However, animation is not always possible and in most cases it can only provide a partial validation.
• *The verification of major safety properties of the system.* Showing that safety critical requirements are satisfied is essential for increasing our confidence in the specifications. There is no alternative to formal proof for performing such verification. Other validation techniques such as review, testing, or animation cannot offer the same degree of confidence.

In general, the proof process was essential for analyzing the complex interactions between the two channels and the external components, and for comprehending the global system. Failed proofs were always useful by revealing subtle properties of the ADC thereby helping to understand the behavior of the whole system. Identifying the causes of failure often revealed unexpected behavior under exceptional circumstances. Most of the failed proofs were due to special cases we did not anticipate, which were caused by complex combinations of events in the environment and in the controller.

Obviously, identifying the cause of a proof failure is a hard problem which cannot be solved mechanically. In theory, failing to prove a conjecture does not necessarily mean that it is false. In practice, however, all our proofs failed by ending in a visibly unprovable goal from which we could exhibit a counter-example. This either convinced us that our belief in the conjecture was wrong or pointed to a possible error in the specifications. The final diagnosis often required the help of GEC-Marconi's engineers.

B. Tool support

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 specifications and the number of proofs performed, our experiment represents a major effort in formal verification. The experiment has shown that such large scale verification is feasible and has corroborated other reports on industrial uses of PVS, such as [9]. Our verification effort would not have succeeded without effective tool support. Tools are necessary simply to cope with the size of the specifications. They are also essential for supporting the proof process.

The requirements of the case study include numerical computations and logical and temporal constraints. The state space of the system is infinite and the numerical relations are not linear. For reasoning about such systems, fully automatic proof support does not exist and the user has to rely on interactive proof assistants. During the validation phase, the PVS theorem prover was used to refute many incorrect proofs. Much of the verification performed does not rely on any sophisticated argument but usually requires a lengthy and detailed case analysis. In our experience, it was fairly easy to get an "almost correct" proof which failed due to a few obscure cases. There is no doubt that without a tool which forces the examination all the details, some of these bad cases would have gone unnoticed.

Therefore tools are necessary for gaining a high level of confidence in the proofs. For systems such as the ADC, the only available tools are interactive theorem provers. For such tools to be of real help in practice, the low level steps of the proofs must be automated as far as possible. For this purpose, two features of PVS were essential: the decision procedures and the possibility of installing automatic rewrite rules.

On a more general level, the case study showed that flexibility and ease of modification are important practical issues. As already mentioned, a lot of effort was spent failing to prove properties and subsequently trying to identify and correct the causes of failure. This often meant that the specifications had to be changed; assumptions were added, new lemmas introduced, or propositions rewritten. The user must be free to make such modifications without too much effort, in particular, without having to manually reexamine all the proofs. The PVS system keeps scripts of the existing proof attempts which can be edited and re-run. This was extremely useful; usually simple adjustments to these scripts were sufficient after changes in the specifications.

C. Difficulties encountered

The SafeFM case study has confirmed the benefits of formal analysis of high level requirements. It has also shown that formal methods can be applied to realistic, complex systems and that large-scale verification is feasible. However, another lesson of the experiment is that the proof process can be expensive and difficult to estimate. It took us eighteen months to complete the work instead of the six months we had originally planned.

Some of these delays can be attributed to our lack of familiarity with PVS. More experienced users would have undoubtedly completed the formal analysis in less time. Other factors can also explain this time overrun:

- There was little guidance on how to apply the PVS effectively to real-time control applications. A non-negligible part of the work was to write basic PVS theories for supporting the specification approach. These theories were modified and updated several times.
- The absence of general libraries was another problem since we had to spend time in proving simple properties of sets or real numbers which were not present in the prelude.
- The initial requirements document was written from a software engineering point of view. It focused on functionality and contained very little information about the system under control. Early proof attempts showed that this information was crucial but several iterations were required before formulating the correct assumptions and getting precise safety requirements.
- The proofs of the safety properties were a lot harder than we expected. This was largely due to the difference in levels of abstraction between the safety and the software requirements. The safety requirements are global system properties. They are related to the physical system and are expressed in terms of the position of wings and flaps. From the safety perspective, only the externally observable behavior of the controller is relevant. On the other hand, the functional requirements are an explicit description of software components inside the controller and they already depend on implementation decisions (such as the dual channel architecture). There was then a large gap between the two levels of description. The difficulty was relating the output variables of the two channels to the actual position of the wings. The complexity arose from the different elements and layers of interactions involved.

D. Doing better

Some of the difficulties of the case study work were due to the lack of maturity of the PVS prover. In order to reduce the effort involved in formal verification it is necessary to provide general purpose PVS libraries. This necessity has already been identified. The new version of PVS comes with a largely expanded prelude and most of the general properties we had to prove are now present in this prelude. PVS2 also includes new features for creating and handling libraries. However, very few are available at present and developing more libraries is still essential for making verification less labor intensive. For example, we found that a library stating basic properties of continuous functions could have helped in some aspects of the case study verification.

Similarly, developing theories designed to tailor PVS towards specific classes of applications would be an important improvement. In the case of real-time systems similar to our case study, the theories defining clocks and other elementary notions could provide basic support. However, in their existing form these theories are still fragmentary and need to be expanded and completed to offer sufficient help.

The case study requirement analysis was largely an exercise of *a posteriori* verification. The validation consisted of proving properties of software requirements which were already available. There was a large gap between the high level of the safety properties and the comparatively low level of the functional specifications. This was the main source of complexity in the safety proofs. In order to simplify the verification, a solution may be to adopt a constructive approach of gradually developing software specifications from the high level requirements of the whole system. This should allow a more localised form of reasoning, proving the safety properties at the highest level of description and then focusing on the correctness of each refinement.

Whatever the techniques used, formal or not, a general improvement in the validation of software specifications for complex systems would be to address the issue of requirements traceability. In order to analyse and validate software specifications, it is essential to know what assumptions the system designers have made and what part of the safety requirements are the responsibility of the software. Discovering this information was one of the most time consuming activities of the experiment. The whole verification effort would have been much less resource-intensive had we started with precise assumptions about the system under control and with clear safety properties.

VI. Conclusions

The case study experiment has confirmed the potential benefits of formal methods for requirements analysis of safety critical systems. 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 proof. Formal verification can help uncover errors, misunderstandings, or subtle, unexpected properties which could easily escape other means of scrutiny such as review or animation. In the right circumstances, increased confidence in the requirements specification can be obtained by proving safety-related properties. For this purpose, it is crucial to adopt from the start a wide point of view and consider the system as a whole. Precise knowledge of the system under control and of the environment, and a clear specification of the safety requirements are essential. The narrow perspective of the software developer is not sufficient.

Existing formal methods and proof tools are mature enough to be applied to large systems of substantial complexity. The main value of tools is to help identify and correct wrong arguments during the proof process. Specification debugging via proof was the principal activity during requirements validation and a lot was learnt about the case study through failed proofs and corrections.

In spite of the potential benefits, the amount of effort and time spent on the case study may still be seen as an obstacle. However, we believe that most of the difficulties encountered were due to the particular context of the study. As experience is gained in the application of PVS to real-time controllers, the amount of effort required for their verification should decrease. The main issue is for the PVS community to develop and make available libraries. These could provide a background of general notions and results, as well as guidance and support theories for facilitating the description of real-time systems.

Acknowledgements

The work described here would not have been possible without the support of our project partners. Particular thanks are due to our GEC-Marconi colleagues who have had to educate us in avionics systems.

References

[1] "The Procurement of Safety Critical Software in Defence Equipment," 1991, Interim Defence Standard 00-55, Issue1.

[2] "Draft IEC Standard 1508 - Functional Safety: Safety-related systems," April 1995, International Electrotechnical Commission, Technical Committee no. 65, Working Group 9/10 (WG 9/10), IEC 65A.

[3] P. Bradley, L. Shackleton, and V. Stavridou, "The SafeFM project," in *Proc. of Safety Critical Systems Symposium 93*, F. Redmill, Ed. February 1993, pp. 168–176, Springer-Verlag.

[4] 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*, vol. 1, no. 6, pp. 541–545, 1996.

[5] 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*, vol. 21, no. 2, pp. 107–125, February 1995.

[6] 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.

[7] T. Boyce, "SafeFM case study report," Tech. Rep. SafeFM-018-GEC-1, SafeFM project, January 1994.

[8] S. Owre, N. Shankar, and J. M. Rushby, *User Guide for the PVS Specification and Verification System*, Computer Science Lab., SRI International, March 1993.

[9] 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.

[10] M. K. Srivas and S. P. Miller, "Formal Verification of an Avionics Microprocessor," Tech. Rep. SRI-CSL-95-04, SRI International, June 1995.

[11] K.-H. Buth, "Automated Code Generator Verification based on Algebraic Laws," Tech. Rep. ProCoS II Report, Kiel KHB 5/1, University of Kiel, September 1995.

[12] J. Hooman, "Correctness of Real Time Systems by Construction," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*. September 1994, pp. 19–40, Springer-Verlag, LNCS 863.

[13] N. Shankar, "Verification of Real-Time Systems Using PVS," in *Computer Aided Verification, CAV'93*. June-July 1993, Springer-Verlag, LNCS 697.

[14] M.J.C. Gordon and T.F. Melham, *Introduction to HOL. A theorem proving environment for higher order logic*, Cambridge University Press, 1993.

[15] A. P. Ravn, H. Rischel, and K. M. Hansen, "Specifying and verifying requirements of real-time systems," *IEEE Trans. on Software Engineering*, vol. 19, no. 1, pp. 41–55, January 1993.

[16] Z. Chaochen, C. A. R. Hoare, and A. P. Ravn, "A calculus of durations," *Information Processing Letters*, vol. 40, no. 5, pp. 269–276, December 1991.

[17] J. U. Skakkebæk and N. Shankar, "Towards a duration calculus proof assistant in PVS," in *Formal Techniques in Real-time and Fault-Tolerant Systems.* September 1994, Springer-Verlag, LNCS 863.

[18] Z. Chaochen, A. P. Ravn, and M. R. Hansen, "An extended duration calculus for hybrid real-time systems," in *Hybrid Systems*, 1993, number 736 in LNCS, pp. 36–59.

[19] A. P. Ravn, *Design of Embedded Real-Time Computing Systems*, Ph.D. thesis, Technical University of Denmark, Lyngby, Denmark, October 1995.

[20] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The Synchronous Data Flow Programming Language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1321, September 1991.

[21] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire, "Programming Real-Time Applications with SIGNAL," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1321–1336, September 1991.

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

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

[24] J. R. Abrial, M. Lee, D. Neilson, N. Scharbach, and I. Sorensen, "The B method for Large Software. Specification, Design and Coding," in *VDM'91: Volume 2: Tutorials.* 1991, pp. 398–405, Springer-Verlag, LNCS 552.

[25] B. Dutertre, "Case study coherent specifications," Tech. Rep. SafeFM-050-RH-1, SafeFM project, July 1994.

[26] T. Boyce, "Formal techniques in analysis and design," Tech. Rep. SafeFM-027-GEC-2, SafeFM project, July 1994.

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

[28] T. Boyce, "ML animation and test case generation," Tech. Rep. SafeFM-036-GEC-1, SafeFM project, January 1995.

Bruno Dutertre received a DEA (French MSc) and a doctorate in computer science from the university of Rennes 1/IFSIC and an engineering degree from INSA/Rennes. He is currently a temporary lecturer at Queen Mary and Westfield College, University of London. Previously he was research assistant on the SafeFM project. His main research interests are formal methods and high-integrity systems, and the application of logic and theorem proving to software engineering and to security protocols. He also worked on model checking techniques for validating SIGNAL programs. Bruno can be contacted by e-mail: bruno@dcs.qmw.ac.uk.

Victoria Stavridou holds a BSc on Electronic Computer Systems and an MSc in the Assessment of Computer Aided Logic Design, both from the University of Salford, UK, as well as a PhD on Equational Specification and Verification of Digital Systems from the University of Manchester. She is a Reader in Computer Science at Queen Mary and Westfield College, University of London. Her research interests include safety critical systems, formal methods and dependability. She has written extensively in these areas. She can be contacted on victoria@dcs.qmw.ac.uk.