Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS

Sam Owre, John Rushby,
Natarajan Shankar, Friedrich von Henke

Abstract— **PVS is the most recent in a series of verification systems developed at SRI. Its design was strongly influenced, and later refined, by our experiences in developing formal specifications and mechanically checked verifications for the fault-tolerant architecture, algorithms, and implementations of a model "reliable computing platform" (RCP) for life-critical digital flight-control applications, and by a collaborative project to formally verify the design of a commercial avionics processor called AAMP5.**

Several of the formal specifications and verifications performed in support of RCP and AAMP5 are individually of considerable complexity and difficulty. But in order to contribute to the overall goal, it has often been necessary to modify completed verifications to accommodate changed assumptions or requirements, and people other than the original developer have often needed to understand, review, build on, modify, or extract part of an intricate verification.

In this paper, we outline the verifications performed, present the lessons learned, and describe some of the design decisions taken in PVS to better support these large, difficult, iterative, and collaborative verifications.

Keywords— **Byzantine agreement, clock synchronization, fault tolerance, flight control, formal methods, formal specification, hardware verification, theorem proving, verification systems, PVS.**

I. Introduction

We consider the chief benefit of formal methods is that they allow certain questions about computational systems to be reduced to calculation. For these methods to be useful in practice, calculations relevant to problems of substantial scale and complexity must be performed efficiently and reliably. This requires mechanized tools, and the main focus of our research has been the development of tools for formal methods that are sufficiently powerful that they can be applied effectively to problems of intellectual or industrial significance.

This paper outlines a number of verifications performed with our tools on applications related to aircraft flight control and describes their influence on the design of PVS, our latest verification system. The rest of this introductory section describes the problem domain for the formal verifications considered here, and briefly introduces our tools. The verifications performed are described in Section II; the lessons we have learned and their influence on the design of PVS are presented in Section III; brief conclusions are given in Section IV.

A. The Problem Domain: Digital Flight Control Systems

Catastrophic failure of digital flight-control systems for passenger aircraft must be "extremely improbable"; a requirement that can be interpreted as a failure rate of less than 10^{-9} per hour [1, paragraph 10.b]. This must be achieved using electronic devices such as computers and sensors whose individual failure rates are several orders of magnitude worse than the requirement. Thus, extensive redundancy and fault tolerance are needed to provide a computing resource of adequate reliability for flight-control applications. Organization of redundancy and fault-tolerance for ultra-high reliability is a challenging problem: redundancy management can account for half the software in a flight-control system [2] and, if less than perfect, can itself become the *primary* source of system failure [3].

There are many candidate architectures for the ultra-reliable "computing platform" required for flight-control applications, but a general approach based on rational foundations was established in the late 1970s and early 1980s by the SIFT project [4]: several independent computing channels (each having their own processor) operate in approximate synchrony; single source data (such as sensor samples) are distributed to each channel in a manner that is resistant to "Byzantine" faults[1] [5], so that each good channel gets exactly the same input data; all channels run the same application tasks on the same data at approximately the same time and the results are submitted to exact-match majority voting before being sent to the actuators. Failed sensors are dealt with by the sensor-conditioning and diagnosis code that is common to every channel; failed channels are masked by the majority voting of actuator outputs. The original SIFT design suffered from performance problems, but several effective architectures based on this general idea have since been developed,

This work was performed for the National Aeronautics and Space Administration Langley Research Center under contracts NAS1 17067 and NAS1 18969.

The authors are with the Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA; von Henke's main affiliation is Fakultät für Informatik, Universität Ulm, Germany. Email: `Owre|Rushby|Shankar@csl.sri.com`, and `vhenke@informatik.uni-ulm.de`

[1] Strictly, a Byzantine fault-tolerant algorithm is one that makes no assumptions about the behavior of faulty components; it can be thought of as one that tolerates the "worst possible" (i.e., Byzantine) faults. In this sense, *Byzantine* faults are generally considered to be those that display asymmetric symptoms: sending one value to one channel and a different value to another, thereby making it difficult for the receivers to reach a common view. *Symmetric* faults deliver wrong values but do so consistently. *Manifest* faults are those that can be detected by all nonfaulty receivers.

including one (called MAFT) by a manufacturer of flight-control systems [6].

These fault-tolerant architectures must be able to withstand multiple faults, and it can require an excessive amount of redundancy to do this if failed channels are left operating (e.g., seven channels are required to withstand two simultaneously active Byzantine faults). Reconfiguration to remove faulty channels reduces the redundancy required, provided further faults do not arrive before reconfiguration has been completed (e.g., five channels are sufficient to withstand two Byzantine faults if the system can reconfigure between arrival of the first and second faults). However, reconfiguration adds considerable complexity to the design, and can thereby promote design faults that reduce overall reliability.

Experimental data shows that the large majority of faults are *transient* (typically single event upsets caused by cosmic rays, and other passing hazards): the device temporarily goes bad and corrupts data, but then (possibly following a reset interrupt from a watchdog timer) it restores itself to normal operation. The potential for lingering harm remains, however, from the corrupted data that is left behind. This contamination can gradually be purged if the computing channels vote portions of their internal state data periodically and replace their local copies by majority-voted versions. This process provides self-stabilizing *transient recovery*; after a while, an afflicted processor will have completely recovered its health, refreshed its state data, and become a productive member of the community again. The viability of this scheme depends on the recovery rate (which itself depends on the frequency and manner in which state data are refreshed with majority voted copies, and on the pattern of dataflow dependencies among the application tasks) and on the fault arrival rate. Markov modeling shows that a nonreconfigurable architecture with transient recovery can provide fully adequate reliability even under fairly pessimistic assumptions.

We mentioned earlier that the distribution of single-source data must be done in a manner that is resistant to Byzantine faults. The clock synchronization that keeps the channels operating in lock-step must be similarly fault tolerant. Byzantine fault-tolerant algorithms are known for both the sensor distribution and clock synchronization problems, but they suffer from some disadvantages. First, the standard Byzantine fault-tolerant clock-synchronization algorithms do not provide transient recovery: there is no fully analyzed mechanism that allows a temporarily disturbed clock to get back into synchronization with its peers. Second, conventional Byzantine fault-tolerant algorithms treat all faults as Byzantine and therefore tolerate *fewer* simple faults than less sophisticated algorithms. For example, a five-channel system ought to be able to withstand two simultaneous symmetric faults (by ordinary majority voting), and as many as four manifest faults (by simply ignoring the manifestly faulty values). Yet a conventional Byzantine fault-tolerant algorithm is only good for one fault of *any* kind in a five-channel system. To overcome this, the MAFT project introduced the idea of *hybrid* fault models and of algorithms that are maximally resistant to simultaneous combinations of faults of different types [7].

Although the principles just sketched are well understood, fully credible analysis of the necessary algorithms and their implementations (which require a combination of hardware and software), and of their synthesis into a total architecture, has been lacking.[2] In 1989, NASA's Langley Research Center began a program to investigate use of formal methods in the design and analysis of a "reliable computing platform" (RCP) for flight-control applications. We supplied our EHDM and (later) PVS verification systems to NASA Langley, and have collaborated closely with researchers there. The overall goal of the program is to develop mechanically checked formal specifications and verifications for the architecture, algorithms, and implementations of a model RCP that is resilient with respect to a hybrid fault model that includes Byzantine and transient faults.

This is a rather ambitious goal, since the arguments for correctness of some of the individual fault-tolerant algorithms are quite intricate, and their synthesis into an overall architecture is of daunting complexity. Because mechanized verification of algorithms and fault-tolerance arguments of the difficulty we were contemplating had not been attempted before, we did not have the confidence to simply lay out a complete architecture and then start verifying it. Instead, we first isolated some of the key challenges and worked on those in a relatively abstracted form, and then gradually elaborated the analysis, and put some of the pieces together. The process is still far from complete and we expect the program to occupy us for some time to come.[3] Later in the program, the goals expanded to include transfer of formal verification technology to US aerospace companies. As part of this technology transfer, we and NASA established a collaboration with Collins Commercial Avionics to apply formal verification to the hardware design and microcode of an advanced commercial avionics computer called AAMP5. This stressed our tools to their limits and led to further refinements in their implementation.

Before describing the verifications performed with them in more detail, we briefly introduce our tools.

B. Our Verification Systems

EHDM, which first became operational in 1984 [11] but whose development still continues, is a system for the development, management, and analysis of formal specifications and abstract programs that extends a line of development that began with SRI's original Hierarchical Development Methodology (HDM) of the 1970's [12]. EHDM's specifica-

[2] Some aspects of SIFT—which was built for NASA Langley—were subjected to formal verification [8], but the treatment was far from complete.

[3] CLI Inc., and ORA Corporation also participate in the program, using their own tools. Descriptions of some of their work can be found in [9] and [10], respectively. The overall program is not large; it is equivalent to about three full-time staff at NASA, and about one each at CLI, ORA, and SRI.

tion language is a higher-order logic with a rather rich type system that includes predicate subtypes. EHDM provides facilities for grouping related material into parameterized modules and supports a form of hierarchical verification in which the theory described by one set of modules can be shown to interpret that of another; this mechanism is used to demonstrate correctness of implementations, and also the consistency of axiomatizations. EHDM provides a notion of implicit program "state" and supports program verification in a simple subset of Ada. However, these capabilities were not exploited by the verifications described here: all algorithms and computations were described functionally. The EHDM tools include a parser, prettyprinter, typechecker, proof checker, and many browsing and documentation aids, all of which use a customized GNU Emacs as their interface. Its proof checker is built on a decision procedure (due to Shostak [13]) for a combination of ground theories that includes linear arithmetic over both integers and rationals. EHDM's proof-checker is not interactive; it is guided by proof descriptions prepared by the user and included as part of the specification text [14].

Development of PVS, our most recent verification system, started in 1991; it was built as a lightweight prototype for a "next generation" version of EHDM, and in order to explore ideas in interactive proof checking. Our goal was considerably greater productivity in mechanically-supported verification than had been achieved with other systems.

The specification language of PVS is similar to that of EHDM, but has an even richer type system that includes dependent types. However, PVS omits the support for hierarchical verification and for program verification present in EHDM. The PVS theorem prover includes similar decision procedures to EHDM, but provides much additional automation—including an automatic rewriter, and use of Binary Decision Diagrams (BDDs) for propositional simplification—within an interactive environment that uses a sequent calculus presentation [15]. The primitive inference steps of the PVS prover are rather powerful and highly automated, but the selection and composition of those primitive steps into an overall proof is performed interactively in response to commands from the user. Proof steps can be composed into higher-level "strategies" that are similar to the tactics of LCF-style provers [16].

Specifications in EHDM and PVS can be stated constructively using a number of definitional forms that provide conservative extension, or they can be given axiomatically, or a mixture of both styles can be used. The built-in types of EHDM and PVS include the booleans, integers, and rationals; enumerations and uninterpreted types can also be introduced, and compound types can be built using (higher-order) function and record constructors (PVS also provides tuples and recursively-defined abstract data types). Standard theories defined in terms of the basic types are preloaded into both systems; in the case of PVS, for example, these provide sets, lists, trees, a constructive representation of the ordinals up to ϵ_0, and many other useful constructions.

The distinguishing feature of both EHDM and PVS is the tight and mutually supportive integration between their specification languages and theorem provers. For example, the type systems of both languages include features (such as predicate subtypes) that render typechecking algorithmically undecidable: in certain cases, the typechecker needs the services of the theorem prover. Conversely, type predicates provide additional information to the theorem prover and thereby increase the effectiveness of its automation.

It is not easy to directly compare EHDM and PVS with other approaches to formal methods, such as those embodied in the Z [17] and VDM [18] notations, or the Boyer-Moore theorem prover [19], [20], since they are based on very different foundations. The HOL system [21] is based on similar foundations to EHDM and PVS, but its language, proof-checker, and environment are much more austere than those of our systems. Over several years of experimentation, we have found that our specification languages have permitted concise and perspicous treatments of all the examples we have tried, and that the PVS theorem prover, in particular, is a more productive instrument than others we have used. The PVS system is freely available under license from SRI International. It can be obtained by anonymous ftp from `ftp.csl.sri.com/pub/pvs` or via the World Wide Web from `http://www.csl.sri.com/pvs.html`. Prospective users of EHDM should contact the authors for information on its availability.

II. Formal Verifications Performed

In this section we describe some of the verifications performed using EHDM and PVS. We concentrate on those undertaken as part of our work with NASA, since these span several years and have had the greatest impact on the development of EHDM, and the design of PVS. Other areas of applications include real-time systems, where PVS has been used by us [22], [23], and by others working independently [24], to formalize and verify real-time properties.

A. *The Interactive Convergence Clock Synchronization Algorithm*

The first verification we undertook in NASA's program was of Lamport and Melliar-Smith's Interactive Convergence Algorithm (ICA) for Byzantine fault-tolerant clock synchronization. At the time, this was one of the hardest mechanized verifications that had been attempted and we began by simply trying to reproduce the arguments in the journal paper that introduced the algorithm [25]. Eventually, we succeeded, but discovered in the process that the proofs or statements of all but one of the lemmas, and also the proof of the main theorem, were flawed in the journal presentation. In developing our mechanically-checked verification we eliminated the approximations used by Lamport and Melliar-Smith and streamlined the argument. We were able to derive a journal-style presentation from our mechanized verification that is not only more precise than the original, but is simpler, more uniform, and easier to follow [26], [27]. Our mechanized verification in EHDM took us a couple of months to complete and required about 200 lemmas (many of which are concerned with "background knowledge," such as summation and properties of the arithmetic mean, that are assumed in informal presentations).

We have modified our original verification several times. For example, we were unhappy with the large number of axioms required in the first version. Since axioms can introduce inconsistencies, definitions are often to be preferred, but the early version of EHDM lacked the necessary mechanisms. Later, when definitional forms guaranteeing conservative extension were added to EHDM, we were able to eliminate the large majority of our axioms in favor of definitions; the axioms that remain are used to state assumptions about the environment and constraints on parameters—properties that are best treated axiomatically rather than definitionally. Even so, Bill Young of CLI, who repeated our verification using the Boyer-Moore prover [28], found that one of the remaining axioms was unsatisfiable in the case of drift-free clocks. We adopted a repair suggested by him (a substitution of $\leq$ for $<$), and also an improved way to organize the main induction. The defective axiom identified by Bill Young did not introduce an inconsistency; rather, it excluded an intended model. Consistency of axioms can be established by exhibiting a model, and we have since done this for our current specification of ICA using the theory interpretation mechanism of EHDM. However, checking that an axiomatic specification captures all (and only) the intended models does not lend itself to a similarly definitive test: it is really a problem of validation ("does the specification say what is intended?"), and must be examined through the human processes of review, introspection, and the exploration of test cases. Formal verification can assist this validation activity by exposing all assumptions, by sharpening their statements, and by allowing the mechanized examination of test cases.

When we first completed our verification of ICA, we assumed that was the end of the matter—the final step in understanding and documenting the algorithm. Later, we discovered that availability of a mechanically checked specification and verification opened new opportunities for further investigations.

The first of these was initiated by our colleague Erwin Liu, who developed and formally verified in EHDM the design of a hardware circuit to perform part of the clock-synchronization function [29]. During circuit design, it became apparent that one of the assumptions of ICA (namely, that the initial clock corrections are all zero) is very inconvenient to satisfy in an implementation. We explored the conjecture that this assumption is unnecessary by simply eliminating it from the formal specification and re-running all the proofs (which takes about 10 minutes on a Sun SPARCstation 2) in order to see which ones no longer succeeded. We found that the proofs of a few internal lemmas needed to be adjusted, but that the rest of the verification was unaffected.

Another change was stimulated by Dan Palumbo and Lynn Graham of NASA, who built equipment for experimenting with clock-synchronization circuitry and found that the observed worst-case skews were significantly better than predicted [30]. They showed informally that observation and theory could be brought into much closer agreement by changing the modeling of errors in the reading of clock values to use the fact that there is no read-error when a processor reads its own clock. This improvement leads to reductions of 10% to 25% in numerical estimates of the worst-case clock skew [31].

Extending the formal verification of ICA to accommodate Palumbo and Graham's improved bound required development of a body of lemmas concerning finite summations. Clock skew between two nonfaulty processors p and q is influenced by their individual skews with processor r. A bound on the maximum skew between p and q is obtained by summing these contributions over all r. Previously, it had only been necessary to split this summation into two—according to whether processor r is faulty or nonfaulty—and an ad-hoc formal treatment was adequate. For Palumbo and Graham's bound, it was necessary to split the summation further according to whether r is the same as one of p or q, and it seemed better to develop a more general theory of summations.

Availability of this general theory then made it feasible to extend ICA to a hybrid fault model (where the summation needs to be further subdivided according to the different kinds of fault). This was accomplished in just a few days of work, and provides a formally verified algorithm that has significantly better fault tolerance than the original algorithm [31]. Among all clock-synchronization algorithms suitable for architectures of the kind used in digital flight-control applications, this hybrid variant of ICA seems to provide the most robust fault tolerance for a given level of redundancy.

B. Other Clock Synchronization Algorithms

There are alternatives to ICA that may be easier to implement. Also, there is a choice in formalizations of clock synchronization whether clocks are modeled as functions from "clock time" to "real time" or the reverse. ICA does it the first way, but the other seemed to fit better into the arguments for an overall architecture. Accordingly, we next embarked on a mechanized verification of Schneider's generalized clock-synchronization protocol, which gives a uniform treatment that includes almost all known synchronization algorithms [32], and models clocks in the "real time" to "clock time" direction. As before, we found a number of small errors in the original argument and were able to produce an improved journal-style presentation as well as the mechanically-checked proof [33].

This general verification, undertaken by Shankar using EHDM, depends on 11 constraints that must be satisfied by any specific instantiation of the general theory. Shankar verified that the instantiation that characterizes ICA satisfies these constraints, thereby providing an independent verification of this algorithm. Paul Miner of NASA argued that one of the constraints (called "bounded delay") is often quite difficult to establish for a given instantiation—

almost as difficult as proving synchronization in the first place. Furthermore, by modifying Shankar's treatment, he was able to verify that this condition could be established once and for all from suitably modified versions of the other 10 constraints [34]. Using this simplified approach, he then formally specified and verified the instantiation that characterizes the very attractive Welch-Lynch fault-tolerant mid-point algorithm [35].

Miner and colleagues at Indiana University later developed and implemented a verified clock synchronization circuit that implements this algorithm [36]. Several formal methods were used in their design and implementation. The register transfer level architecture was developed using the Digital Design Derivation (DDD) system from Indiana [37]: this transforms an abstract state machine specification into a concrete architecture by applying a number of correctness-preserving transformations. Most of the transformations used were built-in to DDD, but some additional *ad-hoc* transformations were required; these were justified using PVS. The register transfer level description was then transformed into a gate-level description using a combination of standard "projections" provided by DDD and custom projections justified in PVS. Finally, a BDD-based tautology checker was used to establish equivalence between some of the combinational circuits obtained by projection and more efficient ones available in a standard library.

Miner has also shown that a slightly adjusted version of the Welch-Lynch midpoint algorithm provides self-stabilizing transient recovery [34, chapter 6]. Formal verification of the general form of this extension is a significant challenge for the future.

C. Byzantine Agreement

Turning from fault-tolerant clock synchronization to sensor distribution, we next focussed on the "Oral Messages" algorithm for Interactive Consistency [38].[4] Bill Bevier and Bill Young at CLI, who had already verified this algorithm, found it "a fairly difficult exercise in mechanical theorem proving" [39]. We suspected that their treatment was more complex than necessary, and attempted an independent verification. We were able to complete this in less than a week, and found that one of the keys to simplifying the argument was to focus on the symmetric formulation (which is actually the form required), rather than the asymmetric Byzantine Generals form [40].

Because of its manageable size and complexity (it is an order of magnitude smaller than the clock-synchronization proofs), we used verification of the Oral Messages algorithm as a test-case in the development of the theorem prover for PVS. Eventually we were able to construct the necessary proofs interactively in under an hour (starting from the specification and a couple of minor lemmas). Thus equipped, we turned to an important variation on the algorithm due to Thambidurai and Park [7] that uses a hybrid fault model, and thereby provides greater fault tolerance than the classical algorithm. Here we found not merely that the journal-style argument for the correctness of the algorithm was flawed, but that the algorithm contained an outright bug. We proposed a modified algorithm and, together with our colleague Pat Lincoln, began to formally verify its correctness—until we found that it, too, was flawed.

The difficulty in discovering the flaw in our modified algorithm was due to an unrelated error in one of the axioms of our specification. The modified algorithm uses a "hybrid majority vote" function (i.e., majority vote with error values excluded). We had axiomatized the properties required of this function, rather than given an implementation, and had got it wrong: we had excluded the values that *should* be manifestly erroneous (i.e., that came from manifestly faulty processors), rather than those that actually were. In other words, our voter was "omniscient." This flaw in the axiomatization masked the bug in our algorithm, so that it was provably "correct," but unimplementable (because the omniscient voter cannot be realized in practice). Pat Lincoln noticed the problem after a couple of days, but this experience underscores the need to be very skeptical of axiomatic specifications and the need to validate them by showing that they are satisfied by some (intended) model. In this case, attempting to show that a specific majority-vote algorithm satisfied our axioms would have revealed the error.

Once we had identified our mistake, we were able to repair it and to develop and formally verify a new and correct algorithm for Interactive Consistency under a hybrid fault model [41], [42]. And this time, we did verify an implementation (a modified version of the Boyer-Moore linear-time MJRTY algorithm [43]) against our axioms for a hybrid majority voter [44]. Overall, this work took less than two weeks, and was primarily undertaken by Pat Lincoln as his first exercise in mechanized formal verification using PVS.

As with clock synchronization, availability of a formally verified algorithm for hybrid interactive consistency created new opportunities for the rapid and reliable exploration of variations. One of these is a version of the algorithm for the architecture of the "Fault Tolerant Processor" (FTP) developed at the C. S. Draper Laboratory [45], [46]. A fundamental result regarding Interactive Consistency states that at least $3n+1$ processors are required to withstand n simultaneous Byzantine faults [5]; thus, in particular, four processors are required to withstand a single fault. Traditionally, the fault-tolerant architecture is symmetrical, and all four processors are identical. The FTP architecture breaks with this tradition and uses only three full processors, plus three much simpler "interstages" whose only function is to relay messages between the main processors.

The FTP architecture is an attractive alternative to a conventional four-plex, since is should be cheaper and more reliable (cost and fault arrival rate are largely determined by the number of processors). Published accounts, how-

[4] Interactive consistency is the problem of distributing consistent values to multiple channels in the presence of faults [5]. It is the symmetric version of the Byzantine Generals problem, and should not be confused with interactive convergence, which is an algorithm for clock synchronization.

ever, do not provide a full description and analysis of the interactive consistency algorithm for FTP. Building on our previous treatments of interactive consistency algorithms, we were able to develop a formally specified and verified algorithm for FTP in a matter of days. This algorithm not only extends the analysis to allow some processors to lack interstages (as may arise following reconfiguration), but also employs a hybrid fault model [47] to withstand a wider range of fault behaviors.

D. Hierarchical Verification of the Reliable Computing Platform

Clock synchronization and interactive convergence are key algorithms in the architecture for the Reliable Computing Platform (RCP), but it is the argument for fault tolerance and transient recovery of the overall architecture that is the main challenge.

A model for the overall architecture of RCP, and the argument for its correctness, were developed by Rick Butler, Jim Caldwell and Ben Di Vito at NASA. Their model and verification were formal, in the style of a traditional presentation of a mathematical argument [48]. Working in parallel, we developed a formal specification and verification of a slightly simplified, but also rather more general model [49]. Before formally specifying and verifying our model in EHDM, we developed a description and proof with pencil and paper. This description was developed with specification in EHDM in mind; it was built from straightforward mathematical concepts and was transliterated more or less directly into EHDM in a matter of hours. The formal verification took about three weeks of part-time work. Some of this time was required because the formal verification establishes a number of subsidiary results that were glossed over in the pencil and paper version, and some of it was required because EHDM's theorem prover lacked a rewriter at that time. However, the mechanically verified theorem is also stronger than the pencil and paper version. The stronger theorem requires a proof by Noetherian induction (as opposed to simple induction for the weaker theorem), which is rather tricky to state and carry out in semi-formal notation, but no more difficult than simple induction in a mechanized setting.

The property established by this verification is that, subject to specified assumptions, a replicated collection of processors using majority voting to provide fault masking and transient recovery can achieve the same external behavior in the presence of faults as a single ideal processor that suffers no faults. The assumptions include those of synchronization, interactive consistency for sensor input, and fault containment. (The last of these requires that damage to state data cannot propagate from one application task to another in a nonfaulty processor; in practice it is achieved using memory management hardware).

The most ambitious formal verification carried out in the program so far was performed by Rick Butler and Ben Di Vito at NASA: it elaborates the two-level model described above into a six-level hierarchy that connects the ideal fault-free single processor all the way down to the details of task management, interprocessor communication, and memory management [50], [51]. The topmost level is called the uniprocessor synchronous (US) model: it is essentially the correctness criterion—a single computer that never fails. The level below this is the replicated synchronous (RS) model, which is similar to the fault-masking model described above; below this is the distributed synchronous (DS) model, which introduces the fact that communication between channels takes time; below this is the distributed asynchronous (DA) model, which connects to the clock synchronization conditions and recognizes that the channels are only approximately synchronized. The DA model relates to both clock synchronization (and thence to the various verifications of clock synchronization algorithms and their implementations) and to a minimal voting model (DA_minv). DA_minv, in turn, is supported by the local executive (LE) model, which introduces details of task management, memory management, and interprocessor communication.

The chain of argument that connect the US and LE models has been formally verified at NASA using EHDM's capabilities for hierarchical verification. The US to RS verification is similar to ours, the others are novel. Overall, this formal specification and verification took several man-years to develop, and is the largest such effort undertaken in EHDM; it is also one of the largest and most elaborate formally verified hierarchical developments known to us. The specification and proof directives (which are part of the specification text in EHDM) are 13,559 lines long; about 4 hours are required to check all 781 proofs on a SPARCstation 10.

E. Experimental Verification of the AAMP5

In 1992, SRI and NASA began a collaborative project with Collins Commercial Avionics to determine whether the formal verification technology we had developed could be applied to a project of industrial scale and in an industrial setting. The chosen project was to apply formal specification and verification to selected parts of an avionics processor under development at Collins called the AAMP5 [52]. The exercise was run as a "shadow" project, not on the critical path of the main development. Processor verification is a very different problem to the algorithm and architecture verifications described above; it was chosen because it provided a well-defined project that was of interest and potential value to Collins.

The AAMP5 is a member of Collins' "Advanced Architecture Microprocessor" (AAMP) family [53]. It is intended to be instruction-set compatible with the earlier AAMP2, which is used in numerous applications for civil aircraft (e.g., there are thirty AAMP2s on board each 747-400), but four times faster. AAMP5 is intended for critical applications such as avionics displays, but not for ultra-critical systems such as autoland or fly-by-wire.

Given formal specifications for a processor at the user-visible, machine code, level (the macro-architecture), and at the implementation, register-transfer, level (the micro-architecture), the task of processor verification is to show that the latter implements the former. In a microcoded processor such as the AAMP5, the micro-architecture is driven by a program (the microcode), rather than being hard-wired, and the verification problem becomes the task of showing that the microcode executing on the micro-architecture implements the macro-architecture.

Microcode verification is not new: it was pioneered by Bill Carter at IBM in the 1970's and applied to elements of NASA's Standard Spaceborne Computer [54]; in the 1980's a group at the Aerospace Corporation verified microcode for an implementation of the C/30 switching computer using a verification system called SDVS [55]; and a group at Inmos in the UK established correctness across two levels of description (in Occam) of the microcode for the T800 floating point unit using mechanized transformations [56]. Similarly, several groups have performed automated verification of nonmicrocoded processors, of which Warren Hunt's FM8501 [57] (and subsequent FM9001 [58]) are among the most substantial. However, none of these previous efforts approaches the scale and complexity of the AAMP5.

Both the macro and micro-architectures of the AAMP5 are complex. The macro-architecture is a stack machine with a large and elaborate instruction set: instructions are variable length, they operate on multiple data types, and among the 209 different instructions are some that provide services normally delivered by the run-time system of a compiler (e.g., procedure state saving, parameter passing, return linkage and reentrancy) and others that provide functions normally associated with an operating system kernel (e.g., interrupt handling, task state saving, context switching). The AAMP5 provides separate executive and user address spaces, as well as separate code and data environments. The micro-architecture has four independent units: bus interface, instruction cache, look-ahead fetch unit, and data processing unit. The latter is microcoded, caches the top elements of the stack in registers, and has a three-stage pipeline. There are approximately 500,000 transistors in the AAMP5 implementation.

Specification and verification of the AAMP5 were undertaken in PVS, which was just entering beta-test as the project started. The project began with Srivas at SRI developing formal specifications for a large part of the macro-architecture (based on the AAMP2 programmer's reference manual). Engineers at Collins reviewed these specifications. Two previously undetected errors in the microcode were discovered during this process, as formal specification forced consideration of the intended behavior of the AAMP5 in some unusual circumstances.

After Srivas had completed a first version of the macro-architecture specification, it was taken over and revised and extended by staff at Collins. In order to validate the specifications, Collins engineers familiar with AAMP5 subjected the PVS specifications to Fagan-style inspections [59] and detected 28 major specification faults (i.e., those affecting correctness) in the process. Conversely, formal specification of the macro-architecture led to clarification or correction of several descriptions in the reference manual, and to reconsideration of the manner in which stack overflow is handled. The engineers who performed the inspections required surprisingly little training in PVS, but considerable work was required to render the specifications in a style they found acceptable (e.g., consistent naming, careful formatting, informative comments).

Parenthetically, we should note that we had at first assumed that the Collins engineers should be introduced to PVS in a "bottom-up" manner, starting with the bitvector library (developed by Rick Butler of NASA Langley)—the rationale being that we would start with the basics and show them how things were built up from very simple beginnings. The effect surprised us: the engineers were appalled ("Isn't this stuff built-in?" was a typical comment) and quite uninterested, for they wanted to focus on the big picture. This episode caused us to drop the bottom-up introduction to PVS: subsequently, new members of the project first encountered PVS by sitting in on inspections—an approach that seems to have been quite successful.

While Collins engineers were validating and extending the specification of the macro-architecture, Srivas at SRI developed a formal specification of the micro-architecture. The specification focussed on the data processing unit (DPU); the other units were described only in terms of their external interactions with the DPU.

When Srivas had finished with the specification of the micro-architecture, it was sent to Collins for revision and inspection, and Srivas focussed on developing proof strategies for the verification. To reduce its scope to a manageable size, the project called for only 13 instructions (each representing a different class) to be formally specified and verified by SRI, with an additional 11 to be undertaken at Collins. However, it turned out that in order to specify the basic 13 operations, the complete micro-architecture and almost the complete macro-architecture had to be specified. (In the end, 108 instructions were formally specified at the macro level; the macro-architecture specification is 2,550 lines of PVS in 48 theories, the micro-architecture specification is 2,679 lines in 20 theories, and the bitvector library is 2,030 lines in 31 theories.) Similarly, in order to verify the basic 13 instructions with tolerable efficiency, and in order to render the process sufficiently systematic that it could be transferred to Collins, it was found necessary to explore a number of different techniques and to develop a collection of reusable proof strategies.

Because axiomatic specifications had proved error-prone in the past, specification of the macro-architecture used a definitional style. This style proved cumbersome for verification, however, so a collection of lemmas in the form of conditional equations was created to represent this information in a form suitable for automated rewriting. The need for fast rewriting and the ability to deal with large propositional structures led to a number of modifications to PVS during this effort. These included caching rewrites and the corresponding state of the congruence closure data structure, and the use of BDDs for propositional simplification. These modifications allowed reasonably efficient verification of the AAMP5 microcode and micro-architecture. Currently, 11 instructions have been verified (several of these are outside the core set of 13) from three instruction classes. Verification of a new instruction class takes about a week, and verification of a new instruction in a known class takes less than a day.

Formal verification revealed several errors in the specifications of the macro and micro-architectures that had not been detected by inspections. Furthermore, it led to the detection of two subtle errors that had been seeded in the microcode (these were different errors than those discovered during formal specification). Because the AAMP5 was a development of an architecture that was very familiar to Collins engineers, and for which they had extensive simulation and diagnostic tools, there was concern that there might be no "natural" bugs in its microcode, and therefore the project might be unable to demonstrate the potential of formal verification to detect bugs. Consequently, with the acquiescence of SRI management, Collins engineers planted a bug in the microcode of one of the instructions in the core set. This bug was of a kind that had proved hard to detect with conventional V&V practices. In addition, a bug that had eluded these conventional practices and made it into an early fabrication of the AAMP5 (where it was discovered in tests of applications code) was left in the microcode supplied to SRI. Formal verification easily exposed both these bugs; furthermore, Srivas was able to extract information from the failed proof attempts that allowed him to describe the necessary microcode corrections to the Collins engineers.

This project demonstrated the feasibility of applying formal verification to a commercial microprocessor [52], but its cost was high (about 3 man-years). However, we attribute much of this to one-time start-up costs and are now planning a follow-on project where the understanding and infrastructure developed for AAMP5 will be applied to a new member of the AAMP family. For the follow-on, formal methods will be used in the design loop and PVS specifications will be the main design documents. We expect that this will reduce overall development time and costs. We also anticipate that improved speed and automation in new versions of PVS will make complete formal verification of the processor cost-competitive with the traditional assurance processes, while providing much greater coverage.

III. Lessons Learned and Their Influence on the Design of PVS

We summarize here some of the main characteristics observed and conclusions drawn from the verifications described above. First, most of the proofs we have been interested in checking, not to mention many of the theorems and some of the algorithms, were incorrect when we started. Thus, we find it at least as important that a verification system should assist in the early detection of error as that it should confirm truth. Second, our axiomatizations were occasionally unsound, and sometimes they were sound but didn't say what we thought they did. Mechanisms for establishing soundness of axiomatizations are clearly desirable (purely definitional specifications are often too restricting), as are techniques for reviewing the content of formal specifications. Third, our verifications are seldom finished: changed assumptions and requirements, the desire to improve an argument or a bound, and simple experimentation, have led us to revise some of our verifications several times. We believe that investment in an existing verification should assist, not discourage, discovery of simplifications, improvements, and generalizations. But this means that the method of theorem proving must be robust in the face of reasonably small changes to the specification. Fourth, our formal specifications and verifications were often used by someone other than their original developer. These secondary users sometimes carry off just a few theories (or ideas) for their own work, sometimes they substantially modify or extend the existing verification, and sometimes they build on top of it; in all cases, they need to understand the original verification. These activities argue for specifications and proofs that are structured or modularized in some way, and that are sufficiently perspicuous that users other than the original authors can comprehend them well enough to make effective use of them.

Finally, our specifications and verifications have often been quite large or complex, and the efficiency of their construction is of vital concern. The most important measure of efficiency is the amount of human time expended in the process from start to finish. For many of our specifications, we found that substantial effort was expended on the formal development of "background theories" such as summations, bitvectors, finite sets, and so on. Clearly, it is necessary that such theories should be made available in libraries for future reuse. Much effort was also spent in debugging specifications: methods such as direct execution or simulation of specifications (sometimes called "animation"), and state exploration or model checking can help explore and validate specifications before full verification is undertaken. In our full verifications, we noted two attributes that are important for overall efficiency: for those verifications (such as the fault tolerant algorithms) whose construction requires significant insight at a number of steps, the main requirement is that the theorem prover should automate the straightforward parts of the proof so that the human user is free to focus on the significant steps; for those verifications that are conceptually straightforward, but large (such as AAMP5), a vital requirement is for raw speed in the basic steps of rewriting, arithmetic, and propositional calculus.

In the following subsections we expand on these points and describe some of the design decisions taken in our languages, support tools, and theorem provers, in light of these experiences.

A. Specification Language

In this section we describe some of the choices made in the design of our specification languages, and discuss some of the changes we have made in the light of experience. The main constraints informing our design decisions have been the desire for a language that is powerfully expressive, yet that nonspecialists find comfortable, that has a straightforward semantics, and that can be given effective mechanized support: this includes very stringent (and early) detection of specification errors, as well as powerful theorem proving.

The domain of problems that we have investigated involves asynchronous communication, distributed execution, real-time properties, fault tolerance, and hierarchical development. One question that arises is the degree of support for these topics that should be built-in to the specification language and its verification system. Our viewpoint here is pragmatic rather than philosophical: we have found that a classical higher-order logic is adequate for formalizing the concepts of interest to us in a perspicuous and effective way. We have also found that the computational aspects of the systems of interest to us are adequately modeled in a functional style and we have not found it necessary to employ Hoare logic or other machinery for reasoning about imperative programs. In part, this is because we have concentrated on verifying algorithms and architectural designs, rather than programs; we have chosen to do so because the available evidence points to these and other early lifecycle concerns (particularly requirements) as the principle sources of failure in safety-critical systems [60].[5]

EHDM does provide a notion of "state" that allows systems to be modeled using state-dependent objects and procedural state transformations; it also provides direct support for reasoning about them in a Hoare logic. We have used this capability in other applications, but even then we have generally found it most convenient to develop the bulk of the specification and verification in a functional style, and to transfer to the imperative style only in the final steps—very much in the manner advocated by Guttag and Horning [62].

We have also used specialized formalisms, such as temporal logic, when they seem appropriate, but we have done so by formalizing them within higher-order logic (see, for example, our embedding of the Duration Calculus [23], and Hooman's treatment of a real-time Hoare calculus [24]). The advantage of embedding such formalisms within a single logic is that it is then easier to combine them, and easier to share common theories such as datatypes, arithmetic, and other prerequisite mathematics. Furthermore, we are not restricted to a fixed selection of formalisms, but can develop specialized notations to suit the problem at hand—rather in the way that productive pencil and paper mathematics is done.

In the case of the examples considered here, it was relatively straightforward to describe the necessary concepts directly within higher-order logic in a manner that reproduced the presentation in standard journal treatments of the topics concerned [25], [32] fairly closely, or that followed a style that had proved comfortable in earlier pencil and paper development (e.g., compare the pencil and paper development of a fault-masking model [48] with a fully formal version [49]). However, allowing these formal specifications to be rendered in a natural syntactic form demands some sophistication of the support tools. For example, the definition of the function *working* that appears in the fault-masking verification [49] was given in EHDM as

$$\begin{array}{l} \mathit{working} : \text{function}[C \to \text{function}[R \to \mathit{bool}]] = \\ \quad (\lambda c : (\lambda r : \mathit{OK}(r)(c))) \end{array}$$

whereas the later PVS allows

$$\mathit{working}(c)\colon \mathit{set}[R] = \{r \mid c \in \mathit{OK}(r)\}.\ ^{6}$$

Here we are exploiting the fact that the axiom of comprehension is sound in higher-order logic, so that sets can be identified with predicates (which are themselves just functions with range type *bool*). The value of syntactic conveniences such as these should not be underestimated; we find that they reduce learning time, and ease comprehension and communication. Readers who are new to the languages can quickly—and correctly—interpret specifications written by others when the notation is close to traditional practice.

Several conveniences that appear syntactic actually require semantic treatment. For example, we allow the propositional connectives such as "or" and the arithmetic and relational operators such as + and $\leq$ to be overloaded with new definitions (while retaining their standard ones). This allows the propositional connectives to be "lifted" to temporal formulas (represented as predicates on the natural numbers), for example, so that if x and y are temporal formulas, $x \vee y$ could be defined to denote their pointwise disjunction. These usages correspond to informal mathematical practice, but their mechanized analysis requires rather powerful strategies for type inference and name resolution.

Just as the syntactic aspects of our languages have been enriched over the years, so have their semantic attributes—and in particular the type systems. Initially we had just the "ground" types (i.e., uninterpreted types, the booleans, and the integer and rational numbers) and the (higher-order) function type constructor. We soon found it convenient to add record and enumeration type constructors, and then—the most significant step of all—predicate subtypes. In PVS we also added tuple types, and dependent type constructions.[7]

As their name suggests, predicate subtypes use a predicate to induce a subtype on some parent type. For example, the natural numbers are specified (in PVS) as:

$$\mathit{nat} : \textbf{type} = \{n : \mathit{int} \mid n \geq 0\}.$$

[5] These systems are developed under stringent controls that are very effective at detecting and eliminating faults introduced in the later lifecycle phases of detailed design and coding. For example, Lutz [61] reports on 197 critical software faults detected during integration and system testing of the Voyager and Galileo spacecraft. Only 3 of these faults were programming errors.

[6] These specification fragments appear here as typeset by the LaTeX-prettyprinters of the two systems. The actual input format is a linear ascii representation. For example, the PVS text is entered as `working(c): set[R] = { r | member(c, OK(r)) }`. The table-driven LaTeX-prettyprinters allow transformations such as that from the prefix function application `member` to the infix $\in$ to be specified easily.

[7] A rather useful dependent construction has been available in EHDM since the beginning through the mechanism of module parameters.

More interestingly, the signature for the division operation (on the rationals) is specified by

$$/\colon [\mathit{rational}, \mathit{nonzero\_rational} \to \mathit{rational}]$$

where

$$\mathit{nonzero\_rational} : \textbf{type} = \{x\colon \mathit{rational} \mid x \neq 0\}$$

specifies the nonzero rational numbers. This constrains division to nonzero divisors, so that a formula such as

$$x \neq y \supset (y - x)/(x - y) < 0$$

requires the typechecker to discharge the proof obligation, or Type-Correctness Condition (TCC),

$$x \neq y \supset (x - y) \neq 0$$

in order to ensure that the occurrence of division is well-typed. Notice that the "context" ($x \neq y$) of the division under a left to right reading appears as an antecedent in the proof obligation. TCCs of this kind establish that the value of the original expression does not depend upon the value of a type-incorrect term; they are generated whenever a term of the parent type appears where one of a predicate subtype is required. The automated procedures of our theorem provers generally dispose of such proof obligations instantly (if they are true!), and the user usually need not be aware of them. This use of predicate subtypes allows certain functions (such as division) that are partial in some other treatments to remain total, thereby avoiding the need for logics of partial terms or three-valued logics.

Related constructions allow nice treatments of errors, such as *pop*(*empty*) in the theory of stacks. Here we can type and axiomatize the stack operations as follows:

$$\begin{array}{l}
\mathit{stack}\colon \textbf{type} \\
\mathit{empty}\colon \mathit{stack} \\
\mathit{nonempty\_stack}\colon \textbf{type} = \{s\colon \mathit{stack} \mid s \neq \mathit{empty}\} \\
\\
\mathit{push}\colon [\mathit{elem}, \mathit{stack} \to \mathit{nonempty\_stack}] \\
\mathit{pop}\colon [\mathit{nonempty\_stack} \to \mathit{stack}] \\
\mathit{top}\colon [\mathit{nonempty\_stack} \to \mathit{elem}] \\
\\
\mathit{pop\_push}\colon \textbf{axiom}\ \mathit{pop}(\mathit{push}(y\colon \mathit{elem}, s\colon \mathit{stack})) = s \\
\mathit{top\_push} : \textbf{axiom}\ \mathit{top}(\mathit{push}(y\colon \mathit{elem}, s\colon \mathit{stack})) = e
\end{array}$$

so that *nonempty_stack* is a predicate subtype of *stack*. With these signatures, the expression *pop*(*empty*) is rejected during typechecking (because *pop* requires a *nonempty_stack* as its argument), and the theorem

$$\mathit{push}(e, s) \neq \mathit{empty}$$

is an immediate consequence of the type definitions. By similar reasoning, the axiom *pop_push* is immediately seen to be type-correct, but type-correctness of the expression

$$\mathit{pop}(\mathit{pop}(\mathit{push}(x, \mathit{push}(y, s)))) = s, \qquad (1)$$

cannot be deduced by such simple syntactic analysis (because the outermost *pop* requires a *nonempty_stack*, but is given the result of another *pop*—which is only known to be a *stack*). However, this expression can be shown to be well-typed by proving the theorem

$$\mathit{pop}(\mathit{push}(x, \mathit{push}(y, s))) \neq \mathit{empty},$$

in order to establish that the argument to the first *pop* is, in fact, a *nonempty_stack*. EHDM and PVS automatically generate this theorem as a proof obligation (i.e., TCC) when typechecking the expression (1), and can prove it easily.

In practice, we would not specify the *stack* data type by means of an explicit set of function signatures and axioms; we would use the *datatype* mechanism of PVS instead. This mechanism (which is related to the "shells" of the Boyer-Moore prover and the "free types" of Z) allows recursively structured data types to be specified very compactly in terms of the relationships among their constructors, accessors, and subtype recognizers. As a datatype, *stack* is specified as follows.

$$\begin{array}{l}
\mathit{stack}[t\colon \textbf{type}]\colon \textbf{datatype} \\
\textbf{begin} \\
\quad \mathit{empty}\colon \mathit{empty\_stack} \\
\quad \mathit{push}(\mathit{top}\colon t, \mathit{pop}\colon \mathit{stack})\colon \mathit{nonempty\_stack} \\
\textbf{end}\ \mathit{stack}
\end{array}$$

Here, *empty* and *push* are the constructors, with corresponding recognizers (and predicate subtypes) *empty_stack* and *nonempty_stack*, and *top* and *pop* are the accessors. This specification automatically generates a theory containing signatures and axioms similar to those shown earlier, together with a structural induction scheme, subterm ordering predicate, and several other useful axioms and definitions. Furthermore, the theory thus generated is guaranteed to be a conservative extension (in particular, its axioms are sure to be consistent), and the theorem prover is able to provide very effective mechanization for these highly stereotyped constructions. The datatype mechanism is very useful for specifying structures such as lists and trees; furthermore, by defining predicate subtypes on the types so constructed, it is easy to define data structures such as ordered binary search trees [63], or constructive representations of the ordinals.

TCCs that are not discharged automatically by the theorem prover are added to the specification text and can be proved later, under the user's control. Untrue TCCs indicate a type-error in the specification, and have proved a potent method for the early discovery of specification errors. For example, the injections are specified as that subtype of the functions associated with the one-to-one property:

$$\begin{array}{l}
\mathit{injection} : \textbf{type} = \\
\qquad \{f : [t_1 \to t_2] \mid \forall (i, j : t_1)\colon f(i) = f(j) \supset i = j\}
\end{array}$$

(here t_1 and t_2 are type parameters). If we were later to specify the function *square* as an injection from the integers to the naturals by the declaration

$$\mathit{square} : \mathit{injection}[\mathit{int}, \mathit{nat}] = \lambda(x : \mathit{int}) \colon x \times x$$

then the PVS typechecker would require us to show that the body of *square* satisfies the *injection* subtype predicate.[8] That is, it requires the TCC $i^2 = j^2 \supset i = j$ to be proved in order to establish that the *square* function is well-typed. Since this conjecture is untrue (e.g., $2^2 = (-2)^2$ but $2 \neq -2$), we are led to discover a fault in this specification.

Notice how use of predicate subtypes here has automatically led to the generation of proof obligations that might require special-purpose checking tools in other systems. Another example of the utility of predicate subtypes in generating proof obligations arises when modeling a system by means of a state machine. In this style of specification, we first identify the components of the system state; an invariant specifies how the components of the system state are related, and we then specify operations that are required to preserve this relation. With predicate subtypes available, we can use the invariant to induce a subtype on the type of states, and can specify that each operation returns a value of that subtype. Typechecking the specification will then automatically generate the proof obligations necessary to ensure that the operations preserve the invariant.

Although TCCs are very useful in contexts such as those described above, there are other contexts where it is desirable to control their generation. For example, the expression $n + m$, where n and m are of type *nat*, has type *int* (because *nat* is a subtype of *int*, but nothing special is known about $+$ on the *nat*s, so it is interpreted as addition on the *int*s). This has the disadvantage that a TCC will be generated whenever $n + m$ appears in a context where a *nat* is required.

One way around this difficulty is to overload the function $+$ with the following definition

$$+(n, m : \mathit{nat}) \colon \mathit{nat} = n + m. \; ^{9}$$

This will generate the TCC

$$\forall(n, m \colon \mathit{int}) \colon n \geq 0 \wedge m \geq 0 \supset n + m \geq 0 \qquad (2)$$

once and for all, but has the disadvantage that the definition must be expanded every time $+$ applied to *nat*s occurs in a proof. We overcome these difficulties in PVS by providing *type judgments*, which are declarations of the following form

$$\textbf{judgment} + \textbf{has\_type}\ [\mathit{nat}, \mathit{nat} \to \mathit{nat}].$$

This declaration causes the TCC (2) to be generated to verify the **has_type** claim, and thereafter allows the typechecker freely to use the information that $+$ is closed on the *nat*s.

Judgments provide the typechecker with additional information about the type(s) of a function. PVS also allows *conversions*, which are functions that the typechecker may apply to convert terms that are type-incorrect in a given context into terms of an acceptable type. An example will help explain the idea. The purpose of the "jet-select" function of the Space Shuttle On-Orbit Digital Autopilot ("Orbit DAP," another of the flight control applications we have studied under NASA sponsorship), is to select and fire a set of up to three reaction control jets. The selected set can be restricted to various combinations of "primary" and "vernier" jets, such as those whose gas plumes do not extend above the cargo bay. It is natural to model *vernier* and *primary* jets as subtypes of a *jet* type. However, the subtype relationship between the *primary* and *jet* types does not induce a subtype relation between the sets on these types.[10] However, there is a fairly natural function that extends a set of *primary* jets to a set of *jet*s:

$$\mathit{extend}(p : \mathit{set}[\mathit{primary}]) \colon \mathit{set}[\mathit{jet}] = \{j : \mathit{jet} \mid j \in \mathit{primary} \wedge j \in p\}.$$

If we declare *extend* to be a **conversion**, then we can supply a set of *primary* jets in places where a set of *jet*s is expected: PVS will automatically convert the former to the latter by applying the *extend* function. (PVS can display the "converted" form of a specification or proof sequent on request, and issues a warning if a given context has more than one applicable conversion.)

Implicit conversions solve a number of specification problems in a simple and effective manner. For example, a state-dependent integer program "variable" x in a Hoare-sentence specification, or a flexible integer variable v in a temporal logic specification, will generally be modeled as functions from "state" or "time," respectively, to the integers. It is very convenient to be able to "lift" the arithmetic operators to apply directly to state-dependent or temporal terms, so one can write $x + 1$ or $v + 7$ rather than $x(s) + 1$ and $v(t) + 7$, and conversions provide an effective way to do this. (Simple overloading of the $+$ operator can require unnatural constructions, such as the introduction of functions $1(s)$ and $7(t)$.)

Dependent types increase expressive convenience still further. We find them particularly convenient for dealing with functions that would be partial in simpler type systems. The standard "challenge" for treatments of partial functions [64] is the function *subp* on the integers defined by

$$\mathit{subp}(i, j) = \textbf{if}\ i = j\ \textbf{then}\ 0\ \textbf{else}\ \mathit{subp}(i, j + 1) + 1\ \textbf{endif}.$$

This function is undefined if $i < j$ (when $i \geq j, \mathit{subp}(i, j) = i - j$) and it is often argued that if a specification language is to admit such a definition, then it must provide a treatment for partial functions. Fortunately, examples such as

[8] We would also be required to discharge the (true) proof obligation generated by the subtype predicate for *nat*: $\forall(x : \mathit{int}) \colon x \times x \geq 0$.

[9] The $+$ on the right is the built-in $+$ on integers.

[10] Recall that a set of *jets* is represented in PVS as a function of type $[\mathit{jet} \to \mathit{bool}]$; PVS does not extend subtype relations on function domains to the corresponding function types. Some systems do provide this extension—usually in a "contravariant" manner, so that $[\mathit{jet} \to \mathit{bool}]$ is a subtype of $[\mathit{primary} \to \mathit{bool}]$ (i.e., the subtype relation on the functions is the reverse of that on their domains).

these do *not* require partial functions: they can be admitted as total functions on a very precisely specified domain. Dependent types, in which the *type* of one component of a structure depends on the *value* of another, are the key to this. For example, in the language of PVS, *subp* can be specified as follows.

$$\begin{array}{l} \mathit{subp}((i : \mathit{int}),\ (j : \mathit{int} \mid i \geq j)) \colon \ \textbf{recursive} \ \ \mathit{int} = \\ \quad \textbf{if} \ \ i = j \ \ \textbf{then} \ \ 0 \ \ \textbf{else} \ \ \ \mathit{subp}(i,\ j+1) + 1 \ \ \textbf{endif} \\ \quad \textbf{measure}\colon i - j \end{array}$$ [11]

Here, the domain of *subp* is the dependent tuple-type

$$[i\colon \mathit{int}, \{j\colon \mathit{int} \,|\, i \geq j\}]$$

(i.e., the pairs of integers in which the first component is greater than or equal to the second) and the function is total on this domain.

Although dependent types and predicate subtypes generally obviate the need to deal with partial functions, they also allow these to be modeled quite straightforwardly. For example, the partial functions f from type t_1 to type t_2 can be represented by the following dependent record type.

$$\mathit{pfun}\colon \textbf{type} = [\#\ \mathit{dom}\colon \mathit{set}[t_1],\ \ \mathit{fun} : [(\mathit{dom}) \to t_2]\ \#]$$

(The $[\# \ldots \#]$ brackets indicate a record type in PVS, and a set or predicate enclosed in parentheses (here *dom*) indicates the corresponding predicate subtype.) If *pf* is of type *pfun*, then its domain is denoted *dom*(*pf*)[12] and its application to argument x in this domain is denoted *fun*(*pf*)(x).[13] This is a little ugly, but by defining the function

$$\begin{array}{l} \mathit{pfun\_appl}(\mathit{pf})\colon [(\mathit{dom}(\mathit{pf})) \to t_2] = \\ \qquad\qquad (\lambda(x\colon (\mathit{dom}(\mathit{pf})))\colon \mathit{fun}(\mathit{pf})(x)) \end{array}$$

to be a conversion, we can write simply *pf*(x) and the typechecker will perform the necessary expansion to *pfun_appl*(*pf*)(x) (and thence to *fun*(*pf*)(x)).

The earliest versions of EHDM required almost all concepts to be specified axiomatically—thereby raising the possibility of inadvertently introducing inconsistencies. Our decisions to support very powerful type-constructions and to embrace the consequence that theorem-proving can be required during typechecking were motivated by a desire to increase the expressive power of those elements of the language for which we could guarantee conservative extension. On the other hand, we do not wish to exclude axiomatic specifications; these are often the most natural way to specify assumptions about the environment, and top-level requirements. Axioms can be proved consistent by exhibiting a model—a process that is closely related to verification of hierarchical developments.

The established way to demonstrate that one level of specification "implements" the requirements of another is to exhibit an "abstraction" (also called "retrieve") function that induces a homomorphism between the concrete and the abstract specification. The required constructions can easily be specified within our specification languages, but we have found the process to be tedious and error-prone (for example, it is easy to overlook the requirement that the abstraction function be surjective). Accordingly, we have provided mechanized support for hierarchical verification since the earliest versions of EHDM.[14] Our mechanization is based on the notion of *theory interpretations* [65, Section 4.7]; the basic idea is to establish a translation from the types and constants of the "source" or abstract specification to those of a "target" or concrete specification, and to prove that the axioms of the source specification, when translated into the terms of the target specification, become provable theorems of that target specification. The difference between the use of theory interpretation to demonstrate correctness of an implementation and to demonstrate consistency of a specification is that for the latter, the "implementation" does not have to be useful, or realistic, or efficient; it just has to exist.[15]

The basic mechanism of theory interpretation is quite easy to implement: a "mapping" module specifies the connection between a source and a target module by giving a translation from the types and constants of the former to those of the latter, and a "mapped" module of proof obligations is then generated. Special care is needed when the equality relation on a type is interpreted by something other than equality on the corresponding concrete type.[16] This construction requires proof obligations to ensure that the mapped equality is a congruence relation (i.e., has the properties of equivalence and substitutivity).

These straightforward mechanisms have become somewhat embellished over time, as the stress of real use has revealed additional requirements. For example, we originally assumed that source modules would be specified entirely axiomatically. This proved unrealistic: modules generally contain a mixture of axiomatic and definitional constructions, and it is necessary for the mapping mechanism to translate definitions (and theorems) into the terms of the target specification. Next, we found that our users wished to interpret not just single modules, but whole chunks of specification in which both source and target spanned several modules. This is quite straightforward to support, except that care needs to be taken to exclude modules common to both source and target (these often include modules that specify mathematical prerequisites common to both levels). As the size of specifications increases, it becomes necessary to introduce more layers into the hierarchical verification. For example, in demonstrating the

[11] The **measure** clause specifies a function to be used in the termination proof.

[12] PVS uses the notation *dom*(*pf*) rather than the more usual *pf.dom* to indicate record access.

[13] Note that the predicate subtype in the type of *fun*(*pf*) will cause the typechecker to generate a proof obligation requiring demonstration that $x \in \mathit{dom}(\mathit{pf})$ in this context.

[14] PVS does not support this at the moment; we are examining a slightly different approach involving quotient types.

[15] What is demonstrated here is *relative* consistency: the source specification is consistent if the target specification is. Generally, the target specification is one that is specified definitionally, or one for which we have some other good reason to believe in its consistency.

[16] For example, if abstractly specified stacks are implemented by a pair comprising an array and a pointer, then the equality on abstract stacks corresponds to equality of the implementing arrays *up to* the pointer; this is not the standard equality on pairs.

consistency of the axiomatization used to specify assumptions about clocks [27], we have a module *algorithm* that uses (imports) the module *clocks*. An interpretation for *algorithm* will normally generate interpretations for the types and constants in *clocks* as well. But if we have already established an interpretation for *clocks*, we will want the interpretation for *algorithm* to refer to it, not generate a new one. Supporting these requirements in a reasonable way is not difficult once the requirements have been understood. Our experience has been that it takes some real-world use to learn these requirements.

B. Support Tools

The previous few paragraphs have outlined some of the complicating details that must be addressed in the support environment for a specification language that provides a rich type system and theory interpretations. A consequence of the design decision that typechecking can require theorem proving is that the support environments for EHDM and PVS provide a far closer integration between the language analysis and theorem proving components than is usual. We discuss this in more detail in the section on theorem proving. More mundane, but no less important, engineering decisions concern the choice of interface, style of interaction, and functions provided by the support tools.

Some specification environments allow specifications to be expressed directly in terms of mathematical symbols such as ∀, ∃, ⊃, and so on. Although superficially attractive, we have found that the burdens of supporting these conveniences outweigh the benefits, bringing in their wake such menaces to productivity as structure editors and a plethora of mouse and menu selections. In the United States, at least, most scientists and engineers are fast touch-typists, and we find that a conventional program editor provides a more productive environment for rapid interaction than a graphical user interface. Consequently, we have adopted the GNU Emacs editor as our interface, and accepted an ascii representation for our specifications. (A side benefit of this is that it is perfectly feasible to use EHDM and PVS from remote ascii terminals.) Our experience has been that it is the naturalness of its semantic foundation and syntactic expression that determines acceptance of a specification notation, not its lexical representation on the screen. Nonetheless, we have taken care to provide a civilized concrete syntax, a competent prettyprinter and, as noted earlier, a LaTeX-prettyprinter that can produce attractively typeset documents for review and presentation.

We are not opposed to use of mouse and menus for selection of the major functions of our systems, nor to graphical presentation of certain outputs, but we have not considered it worthwhile to divert development effort away from more fundamental capabilities in order to provide these rather costly conveniences. Recently, however, availability of improved development tools has significantly reduced the associated costs, and so the latest versions of PVS do provide pull-down menu selection (using the facilities of GNU Emacs 19), and graphical representation of module dependencies and of proof trees (using Tcl/TK [66]). We are also investigating hypertext for the presentation of documentation and proofs (using Mosaic).

Our specifications have been quite large, typically involving hundreds of distinct identifiers and dozens of separate modules. We have found facilities for cross-referencing and browsing essential to productive development of large specifications and verifications, especially when returning to them after an absence, or when building on the work of others. Browsing is an on-line capability that allows the user to instantly refer to the definition or uses of an identifier; cross-reference listings provide comparable information in a static form suitable for typeset documentation. Graphical presentation of dependencies aids comprehension of the structure of large specifications.

Our specifications and verifications are developed over periods of days or weeks and we have found it imperative that the system record the state of a development (including completed and partial proofs) from one session to the next, so that work can pick up where it left off. We have found it best to record such information continuously (so that not everything will be lost if a machine crashes) and incrementally (so that work is not interrupted while the entire state is saved in a single shot).

We have also found it necessary to support version management and careful analysis of the consequences of changes. Version management is concerned with the control of changes to a formal development (ensuring that two people do not modify a module simultaneously, for example) and with tracking the consequences of changes. EHDM at one time had quite elaborate built-in capabilities for version management, maintenance of shared libraries, and so on. These proved unpopular (users wanted direct access to the underlying files), so we have now arranged matters so that EHDM and PVS monitor, but do not attempt to control, access to specification files. Changes to specification files are detected by examining their write-dates, and internal data structures corresponding to changed files are invalidated. Users who wish to exercise more control over modification to specification files can do so using a standard version control package such as RCS.

Tracking the propagation of changes can be performed at many levels of granularity. At the coarsest level, the state of an entire development can be reset when any part of it is changed; at a finer level, changes can be tracked at the module level; and at the finest level of granularity, they can be tracked at the level of individual declarations and proofs. Once the consequences of changes have been propagated, another choice needs to be made: should the affected parts be reprocessed at once, or only when needed? EHDM originally propagated changes at the module level (so that if a module was changed and its internal data structures invalidated, that invalidation would propagate transitively up the tree of modules). Reprocessing (i.e., typechecking and proving) took place under user control and reconstructed the internal data structures of the entire tree of modules. This proved expensive when large specifications were involved. An unsuccessful proof in a module at the top of a tree of modules might necessitate a change to an axiom in a module at the bottom. Re-typechecking the entire tree could take several minutes, with consequent loss of concentration and productivity. EHDM now propagates the consequences of changes at the level of individual declarations, and re-typechecking is done incrementally and lazily (i.e., only when needed), also at the level of declarations. This requires a far more complex implementation, but the increase in human productivity is enormous, as the user now typically waits only seconds while the relevant consequences of a change are propagated. Because it can take several seconds, or even minutes, to replay a proof, this is done only on request. "Proof-tree analysis" (described below) identifies the state of a proof during an evolving verification.

C. Theorem Proving

Theorem proving in support of fairly difficult or large verifications requires a rather large range of capabilities and attributes on the part of the theorem prover or proof checker. Furthermore, we have found that each formal verification evolves through a succession of phases, not unlike the lifecycle in software development, and that different requirements emerge at different phases. We have identified four phases in the "verification lifecycle" as follows.

Exploration: In the early stages of developing a formal specification and verification, we are chiefly concerned with exploring the best way to approach the chosen problem. Many of the approaches will be flawed, and thus many of the theorems that we attempt to prove will be false. It is precisely in the discovery and isolation of mistakes that formal verification can be of most value. Indeed, the philosopher Lakatos argues similarly for the role of proof in mathematics [67]. According to this view, successful completion is among the least interesting and useful outcomes of a proof attempt at this stage; the real benefit comes from failed proof attempts, since these challenge us to revise our hypotheses, sharpen our statements, and achieve a deeper understanding of our problem: proofs are less instruments of justification than tools of discovery [68].

The fact that many putative theorems are false imposes a novel requirement on theorem proving in support of verification: it is at least as important for the theorem prover to provide assistance in the discovery of error, as that it should be able to prove true theorems with aplomb. Most research on automatic theorem proving has concentrated on proving true theorems; accordingly, few heavily automated provers terminate quickly on false theorems, nor do they return useful information from failed proof attempts. By the same token, powerful heuristic techniques are of questionable value in this phase, since they require the user to figure out whether a failed proof attempt is due to an inadequate heuristic, or a false theorem.

Development: Following the exploration phase, we expect to have a specification that is mostly correct and a body of theorems that are mostly true. Although debugging will still be important, the emphasis in the development phase will be on *efficient* construction of the overall verification. Here we can expect to be dealing with a very large body of theorems spanning a wide range of difficulty. Accordingly, efficient proof construction will require a wide range of capabilities. We would like small or simple theorems to be dealt with automatically. Large and complex theorems will require human control of the proof process, and we would like this control to be as straightforward and direct as possible.

In our experience, formal verification of even a moderately sized example can generate large numbers of lemmas involving arithmetic. Effective automation of arithmetic, that is the ability to instantly discharge formulas such as

$$x \le y \ \wedge \ x \le 1 - y \ \wedge \ 2 \times x \ge 1 \ \supset \ F(2 \times x) = F(1)$$

(where x and y are rational numbers), is therefore essential to productive theorem proving in this context.

Our proof checkers include decision procedures for ground linear arithmetic: that is expressions involving constants, variables, and uninterpreted function symbols, the relations $<$, $>$, $\le$, $\ge$, $=$, and $\neq$, and the arithmetic operations of addition, subtraction, and multiplication, but with multiplication restricted to the linear case (i.e., multiplication by literal constants only). These are true (i.e., complete) decision procedures over the reals, and heuristically effective over the integers (we have only once encountered the integer incompleteness in practice) [69], [70].

It would, in our view, be quite infeasible to undertake verifications that involve large amounts of arithmetic (such as clock synchronization) without arithmetic decision procedures. However, it has also been our experience that seemingly non-arithmetic topics (such as fault masking) require a surprising quantity of elementary arithmetic (for example, inequality chaining, and "+1" arguments in inductions). Verification systems that lack automation of arithmetic and propositional reasoning require their users to waste inordinate amounts of effort establishing trivial facts.

Other common steps in proofs arising from formal verification are to expand the definition of a function and to replace an instance of the left hand side of a (conditional) equation by the corresponding instance of the right hand side. Both operations can be mechanized by the technique known as "rewriting," and the effectiveness of a theorem prover is strongly influenced by the automation and efficiency of its rewriter. A simple rewriter expands definitions and rewrites first-order equations under the user's guidance. A more sophisticated rewriter will also consider conditional equations (those of the form *condition* $\supset$ *lhs* = *rhs*), and may attempt certain higher-order cases (where it is necessary to find instantiations for function variables). Automatic rewriters require less guidance from the user and use heuristics to select which rewrites to apply.

In our experience, the minimum capabilities required for productive theorem proving during the development phase include effective automation for arithmetic and rewriting. However, it is not enough for a prover to have arithmetic and rewriting capabilities that are individually powerful: these two capabilities need to be tightly integrated. For example, the arithmetic procedures must be capable of invoking rewriting for simplification—and the rewriter should employ the arithmetic procedures in discharging the conditions of a conditional equation, or in simplifying expanded definitions by eliminating irrelevant cases. Theorem provers that are productive in verification systems derive much of their effectiveness from tight integration of powerful primitives such as rewriting and arithmetic decision procedures—and the real skill in developing such provers is in constructing these integrations [71]. More visibly impressive capabilities such as the automation of proof by induction are useful (and we do provide them), but of much less importance than competence in combining powerful basic inference steps including arithmetic and rewriting.

An integrated collection of highly effective primitive inference steps is one requirement for productive theorem proving during the proof development phase; another is an effective way for the user to control and guide the prover through larger steps. Even "automatic" theorem provers need some human guidance or control in the construction and checking of proofs. Some receive this guidance indirectly through the order and selection of results they are invited to consider (the Boyer-Moore prover is like this), others in the form of a program that specifies the proof strategy to be used (the "tactics" of LCF-style provers such as HOL [21] are like this). We have found that direct instruction by the user seems the most productive and most easily understood method of guidance, provided the basic repertoire of operations is not too large (no more than a dozen or so). And we find that a style of proof based on Gentzen's Sequent Calculus allows information to be presented to the user in a very compact but understandable form, and also organizes the interaction very conveniently.

A large verification often decomposes into smaller parts that are very similar to each other and we have found it useful to allow the user to specify customized proof "strategies" (similar to LCF-style tacticals) that can automate the repetitive elements of the proof. We have also found strategies very useful for constructing general-purpose proof procedures that are higher-level and more automatic than the primitive proof procedures.

Presentation: Formal verification may be undertaken for a variety of purposes; the "presentation" phase is the one in which the chosen purpose is satisfied. For example, one important purpose is to provide evidence to be considered in certifying that a system is fit for its intended application. We do not believe the mere fact that certain properties have been formally verified should constitute grounds for certification; the *content* of the verification should be examined, and human judgment brought to bear. This means that one product of verification must be a genuine proof—that is a chain of argument that will convince a human reviewer. It is this proof that distills the insight into why a certain design does its job, and it is this proof that we will need to examine if we subsequently wish to change the design or its requirements.

Many powerful theorem-proving techniques work in ways that do not lend themselves to the extraction of a readable proof, and are unattractive on this count. For example, resolution theorem provers do not generate a conventional proof at all, while heuristic methods can generate proofs that follow "unnatural" paths, and low-level proof checkers overwhelm the reader with trivial detail. It seems to us that the most promising route to mechanically-checked proofs that are also readable is to allow the user to indicate the major steps, while routine ones are heavily automated and regarded as atomic.

Other purposes for which verification may be undertaken (for example, to determine the exact assumptions that underlie a certain theorem, or to gain insight into an algorithm) can require abstracting information from a proof, and it is useful if a theorem prover has tools that can present such abstracts. Both EHDM and PVS, for example, can provide a list of all the definitions, axioms, and lemmas referenced in a proof and, recursively, in the proofs of its lemmas. PVS can also display a graphical representation of the proof tree and we are also investigating use of hypertext to assist the active exploration of proofs.

Maintenance and Generalization: Designs are seldom static; user requirements may change with time, as may the interfaces and services provided by systems that interact with the one under study. A verification may therefore need to be revisited periodically in order to adapt to changes. Thus, in addition to the human-readable proof, a second product of formal verification should be a description that guides the theorem prover to repeat the verification without human guidance. This proof description should be robust—describing a strategy rather than a line-by-line argument—so that small changes in the specification of lemmas or assumptions will not derail it. Some degree of automation in the theorem prover seems essential to this capability: it is difficult for proof checkers, which require very detailed guidance, to adapt to even small changes in a specification.

In addition to the modifications and adjustments that may be made to accommodate changes in the original application, another class of modifications—generalizations—may be made in order to support future applications, to distill general principles, or to explore alternative assumptions and designs. Many of the verifications we have performed have been generalizations of earlier ones, and much of the benefit we have derived from formal verification has been in the exploration of changed assumptions and modified algorithms. Investment in formal verification yields most value if its products can be reused.

Consequences for Prover Design

The evolution of our theorem proving systems to best serve the various requirements described above has followed two main tracks: increasingly powerful automation of low-level inference steps, such as arithmetic reasoning and rewriting, and increasingly direct and interactive control by the user for the higher level steps. We have found this combination to provide greater productivity than that achieved either with highly automated provers that must be kept on a short leash, or with low level proof checkers that must be dragged towards a proof.

One of the greatest advantages provided by interactive theorem provers is the ability to back out of (i.e., undo) unproductive lines of exploration. This can often save much work in the long run: if a case-split is performed too soon, then many identical subproofs may be performed on each of the branches. A user who recognizes this can back up to before the case-split, do a little more work there so that the offending subproof is dealt with once and for all, and then invoke the case-split once more.

In the interests of enhancing productivity for the human user, we have made a number of design decisions that have entailed complex implementation strategies. For example, we allow the user to invent and introduce new lemmas or definitions during an ongoing proof; this flexibility is very valuable, but requires tight integration between the theorem prover and the rest of the verification system: the prover must be able to call the parser and typechecker in order to admit a new definition (and also when substitutions are proposed for quantified variables), and typechecking can then generate further proof obligations.

A yet more daring freedom is the ability to modify the statement of a lemma or definition during an ongoing proof. Much of what happens during a proof attempt is the discovery of inadequacies, oversights, and faults in the specification that is intended to support the theorem. Having to abandon the current proof attempt, correct the problem, and then get back to the previous position in the proof, can be very time consuming. Allowing the underlying specification to be extended and modified during a proof (as we do in PVS) confers enormous gains in productivity, but the mechanisms needed to support this in a sound way are quite complex.

Interactive theorem provers or proof checkers must display the evolving state of a proof so that the user can study it and propose the next step. It is generally much easier for the user to comprehend the proof display if it is expressed in the same terms as the original specification, rather than some canonical or "simplified" form. This means that the external representations of structures need to be maintained along with their internal form. For example, the "let" construct is treated internally as a λ-application, but must be presented to the user as a "let," even after it has undergone transformations such as the expansion of defined terms appearing within it. Obviously, formulas change as proof steps are performed, but it is usually best if each transformation in the displayed proof corresponds to an action explicitly invoked by the user. For example, EHDM always eliminates quantifiers by Skolemization, but for PVS we found it best to retain quantifiers until the user explicitly requests a quantifier-elimination step.

Interactive theorem provers must avoid overwhelming the user with information. Ideally, the user should be expected to examine less than a screenful of information at each interaction. It requires powerful low level automation to prune (only) irrelevant information effectively. For example, irrelevant cases should be silently discarded when expanding definitions—so that expanding a definition of the form

$$f(x) = \textbf{if } x = 0 \textbf{ then } A \textbf{ else } B \textbf{ endif}$$

in the context $f(z+1)$ where z is a natural number should result in simply B. Such automation requires tight integration of rewriting, arithmetic, and the use of type information.

An interactive prover should allow the user to attack the subcases of a proof in any order, and to use lemmas before they have been proved. Often, the user will be most interested in the main line of the proof, and may wish to postpone minor cases and boundary conditions until satisfied that the overall argument is likely to succeed. In these cases, it is necessary to provide a macroscopic "proof-tree analyzer" to make sure that all cases and lemmas are eventually dealt with, and that all proof obligations arising from typechecking are discharged. In addition to this "honesty check," our systems can identify all the axioms, definitions, assumptions and lemmas used in the proof of a formula (and so on recursively, for all the lemmas used in the proof). Such information helps eliminate unnecessary axioms and definitions from theories, and identifies the assumptions that must be validated by external means.

The main difference between the EHDM and PVS theorem provers is that the latter is interactive and incorporates the techniques described above in order to enhance the effectiveness of its interaction. Users told us that they found PVS "at least an order of magnitude" more productive in use than EHDM, and we were, ourselves, quite satisfied with its performance when exploring the main line of difficult proofs. However, we found that it required a little too much interaction when dealing with straightforward lemmas and minor proof branches. We were able to remedy this by defining "strategies" to automate most of the straightforward proofs that we encountered.

PVS provides a simple "strategy language" for combining basic proof procedures into strategies that are akin to the tacticals of LCF-like provers. This can easily lead to a proliferation of rather specialized strategies, however, so as we gained experience we amalgamated many strategies into a few very powerful ones of broad applicability. The functionalities of the primitive proof procedures were adjusted to make them more suitable as building blocks for the higher level strategies. With this arrangement, we find that it is only necessary to remember about a dozen high level strategies, plus the primitive procedures (there are about 20 of them), in order to accomplish proofs using an effective combination of interaction and automation. Examples of higher level strategies provided in PVS are one that establishes properties of recursively-defined functions by induction, and another that is very effective on "obvious" lemmas and proof branches. This strategy sets up all defined functions as automatic rewrite rules, and then iteratively performs Skolemization, rewriting, propositional simplification, heuristic instantiation, and applies arithmetic and other decision procedures. The effectiveness of this strategy is very largely due to the way the rewriter is controlled and to its interaction with the decision procedures. Using rewriting to expand every function definition at every opportunity is seldom effective (and with recursive functions it is nonterminating): many function definitions contain embedded *if-then-else* constructs and expanding these blindly can lead to exponential case-splits. The control technique employed most often by the PVS rewriter will only expand a function definition whose body has a top-level *if-then-else* if the decision procedures are able to simplify the *if* test to either *true* or *false*. This technique generally keeps the rewriter on a productive path, and the case-splits under control.

Our use of strategies may be contrasted to the use of tactics in LCF-style provers such as HOL [21]. Whereas we use powerful primitive inferences and employ strategies to build yet higher-level automation, HOL builds almost everything using tactics, since its built-in proof procedures perform only the elementary inferences of its logic. We doubt that the efficiency required to complete the verifications described here at reasonable cost can be achieved using the HOL approach. The argument advanced in its favor is manifest soundness. It is true that our decision procedures and other powerful primitive inferences have more complicated implementations than the elementary inferences of logic and require (and have received) careful scrutiny to ensure soundness, but most of their complexity is concerned with search, where bugs will affect termination and completeness, not soundness. Users must weigh the arguments and the evidence and make their own choices in these matters.

Although the automation described above proved very effective in our verification of the fault-tolerant algorithms and architectures, it foundered when we applied it to AAMP5 and other large hardware examples. The basic approach still seemed effective, for these hardware examples are very regularly structured and the proofs are conceptually straightforward, but PVS was overwhelmed by their sheer size. Tens of minutes could be spent in performing the large numbers of rewrites required, and the resulting formulas were so large that the simple propositional simplifier used in PVS ran out of space (it would also generate subgoals that were permutations of other subgoals).

We overcame these difficulties using an off-the-shelf BDD package [72] to provide very efficient propositional simplification, and by caching information about rewrites. The cache is a hash-table where, corresponding to a term a, the result of the most recent rewriting of a is kept along with the logical "context" at the time of the rewrite. The context consists of the congruence closure data structure maintained by the decision procedures and the set of rewrite rules current at the time of rewrite. If the term a is encountered again within the same logical context, the result of the rewrite is taken from the cache and the rewriting steps are not repeated. The information that an expression could not be rewritten in a context is also cached (and is probably more heavily used than the information about successful rewriting). Use of BDDs and cached rewrites significantly increases the performance of PVS on hardware examples (for example, they allow the microcode of the benchmark "Tamarack" processor to be verified completely automatically in about five minutes [73]) and these enhancements were instrumental in our ability to undertake mechanized analysis of the AAMP5.

IV. Conclusions

We have described our experiences in developing mechanically-checked formal verifications for several quite difficult and large examples arising in fault-tolerant systems and microprocessor design. As well as ourselves, specifications and verifications were developed by colleagues at SRI who had not been involved in the development of our tools, and by collaborators thousands of miles away at Collins Commercial Avionics and at NASA Langley Research Center. The evolution of our languages and tools in response to the lessons learned from these experiences took us in the direction of increasingly powerful type systems, and increasingly interactive and powerfully automated theorem proving. Powerful type systems allow many constraints to be embedded in the types, so that the main specification is uncluttered and typechecking can provide a very effective consistency check. Effectively automated and user-guided theorem proving also assists the early detection of errors, and the productive development of proofs whose information content can assist in the certification of safety-critical systems [60].

We found that formal verification provides many benefits besides proof of "correctness." These include debugging (i.e., discovery of *in*correctness), complete enumeration of assumptions, sharpened statements of assumptions and lemmas, streamlined arguments, and an enhanced understanding that can lead to further improvements. Furthermore, a formal specification and verification is a reusable intellectual resource that can support reliable and relatively inexpensive exploration of design alternatives and the consequences of changed assumptions or requirements.

Most of the techniques we employ were pioneered by others. For example, Nuprl [74] and Veritas [75] provide predicate subtypes and dependent types; theory interpretations were used in Iota [76]; our theorem proving techniques draw on LCF [16], the Boyer-Moore prover [19], [20], and on earlier work at SRI [13]. Our systems differ from others in tightly integrating capabilities that usually occur separately; this has allowed us to provide expressive specification languages and powerful and very effective mechanization within a classical framework. It should be noted that many of the design choices we have made are tightly coupled: for example, predicate subtypes and dependent types bring great richness of expression to a logic of total functions but require theorem proving to ensure type correctness, which is only feasible if the theorem prover is highly effective; effective theorem proving needs decision procedures for arithmetic and equality over uninterpreted function symbols, and these require that functions are total.

We consider these design choices to have served us well and, at some risk of complacency, we are satisfied with them; although we plan to improve on the details of our languages and mechanizations, and continue to seek major improvements in efficiency, we do not expect to change the main decisions. Direct comparisons with alternative approaches would support objective evaluation, but will not be possible until more verification systems are capable of undertaking mechanically checked verifications of the scale and difficulty described here.

For the future, we are investigating techniques for earlier exploration and validation of specifications, so that the path to eventual confrontation with a theorem prover is made more gradual. The techniques we are examining include direct execution or "animation," state-exploration, and fault-tree analysis. We are also examining ways to extract more useful diagnostic information from failed proof attempts, such as returning counterexamples from the decision procedures. In collaboration with David Dill at Stanford University, we are studying techniques for combining theorem proving with state-exploration and model-checking methods. In this regard, we have developed an experimental translator from Murϕ [77] to PVS, and have connected a BDD-based decision procedure for the modal μ-calculus to PVS, giving us similar capabilities to SMV [78]. We are also exploring more efficient approaches to hardware verification [79], [80] and improved support for requirements specifications in the tabular style advocated by Parnas and others [81], [82].

Acknowledgments:

The work reported here owes a very great deal to our collaborators at NASA Langley Research Center: Rick Butler, Jim Caldwell, Michael Holloway, Paul Miner, and Ben Di Vito, and to Steve Miller at Collins Commercial Avionics. We also thank colleagues at SRI: Pat Lincoln, Erwin Liu, and Mandayam Srivas, who performed several of the verifications mentioned here, and David Cyrluk, Sreeranga Rajan, and Carl Witty, who contributed to the tools development.

References

[1] Federal Aviation Administration, *System Design and Analysis*, June 21, 1988, Advisory Circular 25.1309-1A.

[2] R. W. Dennis and A. D. Hills, "A fault tolerant fly by wire system for maintenance free applications", in *9th AIAA/IEEE Digital Avionics Systems Conference*, Virginia Beach, VA, Oct. 1990, The Institute of Electrical and Electronics Engineers, pp. 11–20.

[3] Dale A. Mackall, "Development and flight test experiences with a flight-crucial digital control system", NASA Technical Paper 2857, NASA Ames Research Center, Dryden Flight Research Facility, Edwards, CA, 1988.

[4] John H. Wensley, Leslie Lamport, Jack Goldberg, Milton W. Green, Karl N. Levitt, P. M. Melliar-Smith, Robert E. Shostak, and Charles B. Weinstock, "SIFT: Design and analysis of a fault-tolerant computer for aircraft control", *Proceedings of the IEEE*, vol. 66, no. 10, pp. 1240–1255, Oct. 1978.

[5] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults", *Journal of the ACM*, vol. 27, no. 2, pp. 228–234, Apr. 1980.

[6] R. M. Kieckhafer, C. J. Walter, A. M. Finn, and P. M. Thambidurai, "The MAFT architecture for distributed fault tolerance", *IEEE Transactions on Computers*, vol. 37, no. 4, pp. 398–405, Apr. 1988.

[7] Philip Thambidurai and You-Keun Park, "Interactive consistency with multiple failure modes", in *7th Symposium on Reliable Distributed Systems*, Columbus, OH, Oct. 1988, IEEE Computer Society, pp. 93–100.

[8] P. M. Melliar-Smith and R. L. Schwartz, "Formal specification and verification of SIFT: A fault-tolerant flight control system", *IEEE Transactions on Computers*, vol. C-31, no. 7, pp. 616–630, July 1982.

[9] William R. Bevier and William D. Young, "Machine checked proofs of the design of a fault-tolerant circuit", *Formal Aspects of Computing*, vol. 4, no. 6A, pp. 755–775, 1992.

[10] Mandayam Srivas and Mark Bickford, "Verification of the Ft-Cayuga fault-tolerant microprocessor system, volume 1: A case-study in theorem prover-based verification", Contractor Report 4381, NASA Langley Research Center, Hampton, VA, July 1991, (Work performed by ORA Corporation).

[11] P. Michael Melliar-Smith and John Rushby, "The Enhanced HDM system for specification and verification", in *Proc. VerkShop III*, Watsonville, CA, Feb. 1985, pp. 41–43, Published as ACM Software Engineering Notes, Vol. 10, No. 4, Aug. 85.

[12] Jay M. Spitzen, Karl N. Levitt, and Lawrence Robinson, "An example of hierarchical design and proof", *Communications of the ACM*, vol. 21, no. 12, pp. 1064–1075, Dec. 1978.

[13] Robert E. Shostak, "Deciding combinations of theories", *Journal of the ACM*, vol. 31, no. 1, pp. 1–12, Jan. 1984.

[14] John Rushby, Friedrich von Henke, and Sam Owre, "An introduction to formal specification and verification using EHDM", Tech. Rep. SRI-CSL-91-2, Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1991.

[15] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A prototype verification system", in *11th International Conference on Automated Deduction (CADE)*, Deepak Kapur, Ed., Saratoga, NY, June 1992, vol. 607 of *Lecture Notes in Artificial Intelligence*, pp. 748–752, Springer-Verlag.

[16] M. Gordon, R. Milner, and C. Wadsworth, *Edinburgh LCF: A Mechanized Logic of Computation*, vol. 78 of *Lecture Notes in Computer Science*, Springer-Verlag, 1979.

[17] J. M. Spivey, Ed., *The Z Notation: A Reference Manual*, Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, second edition, 1993.

[18] Cliff B. Jones, *Systematic Software Development Using VDM*, Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, second edition, 1990.

[19] R. S. Boyer and J S. Moore, *A Computational Logic*, Academic Press, New York, NY, 1979.

[20] R. S. Boyer and J S. Moore, *A Computational Logic Handbook*, Academic Press, New York, NY, 1988.

[21] M. J. C. Gordon and T. F. Melham, Eds., *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*, Cambridge University Press, Cambridge, UK, 1993.

[22] Natarajan Shankar, "Verification of real-time systems using PVS", In Courcoubetis [83], pp. 280–291.

[23] Jens U. Skakkebæk and N. Shankar, "Towards a Duration Calculus proof assistant in PVS", In Langmaack et al. [84], pp. 660–679.

[24] Jozef Hooman, "Correctness of real time systems by construction", In Langmaack et al. [84], pp. 19–40.

[25] L. Lamport and P. M. Melliar-Smith, "Synchronizing clocks in the presence of faults", *Journal of the ACM*, vol. 32, no. 1, pp. 52–78, Jan. 1985.

[26] John Rushby and Friedrich von Henke, "Formal verification of the Interactive Convergence clock synchronization algorithm using EHDM", Tech. Rep. SRI-CSL-89-3R, Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1989 (Revised August 1991), Original version also available as NASA Contractor Report 4239, June 1989.

[27] John Rushby and Friedrich von Henke, "Formal verification of algorithms for critical systems", *IEEE Transactions on Software Engineering*, vol. 19, no. 1, pp. 13–23, Jan. 1993.

[28] William D. Young, "Verifying the Interactive Convergence clock-synchronization algorithm using the Boyer-Moore prover", NASA Contractor Report 189649, NASA Langley Research Center, Hampton, VA, Apr. 1992, (Work performed by Computational Logic Incorporated).

[29] Erwin Liu and John Rushby, "A formally verified module to support Byzantine fault-tolerant clock synchronization", Project report 8200-130, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 1993.

[30] Daniel L. Palumbo and R. Lynn Graham, "Experimental validation of clock synchronization algorithms", NASA Technical Paper 2857, NASA Langley Research Center, Hampton, VA, July 1992.

[31] John Rushby, "A formally verified algorithm for clock synchronization under a hybrid fault model", in *Thirteenth ACM Symposium on Principles of Distributed Computing*, Los Angeles, CA, Aug. 1994, Association for Computing Machinery, pp. 304–313.

[32] Fred B. Schneider, "Understanding protocols for Byzantine clock synchronization", Tech. Rep. 87-859, Department of Computer Science, Cornell University, Ithaca, NY, Aug. 1987.

[33] Natarajan Shankar, "Mechanical verification of a generalized protocol for Byzantine fault-tolerant clock synchronization", In Vytopil [85], pp. 217–236.

[34] Paul S. Miner, "Verification of fault-tolerant clock synchronization systems", NASA Technical Paper 3349, NASA Langley Research Center, Hampton, VA, Nov. 1993.

[35] J. Lundelius Welch and N. Lynch, "A new fault-tolerant algorithm for clock synchronization", *Information and Computation*, vol. 77, no. 1, pp. 1–36, Apr. 1988.

[36] Paul S. Miner, Shyamsundar Pullela, and Steven D. Johnson, "Interaction of formal design systems in the development of a fault-tolerant clock synchronization circuit", in *13th Symposium on Reliable Distributed Systems*, Dana Point, CA, Oct. 1994, IEEE Computer Society, pp. 128–137.

[37] Bhaskar Bose, "DDD—a transformation system for Digital Design Deriviation", Tech. Rep. 331, Computer Science Department, Indiana University, Bloomington, IN, May 1991.

[38] Leslie Lamport, Robert Shostak, and Marshall Pease, "The Byzantine generals problem", *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, July 1982.

[39] W. R. Bevier and W. D. Young, "Machine-checked proofs of a Byzantine agreement algorithm", Tech. Rep. 55, Computational Logic Incorporated, Austin, TX, June 1990.

[40] John Rushby, "Formal verification of an Oral Messages algorithm for interactive consistency", Tech. Rep. SRI-CSL-92-1, Computer Science Laboratory, SRI International, Menlo Park, CA, July 1992, Also available as NASA Contractor Report 189704, October 1992.

[41] Patrick Lincoln and John Rushby, "Formal verification of an algorithm for interactive consistency under a hybrid fault model", In Courcoubetis [83], pp. 292–304.

[42] Patrick Lincoln and John Rushby, "A formally verified algorithm for interactive consistency under a hybrid fault model", in *Fault Tolerant Computing Symposium 23*, Toulouse, France, June 1993, IEEE Computer Society, pp. 402–411.

[43] Robert S. Boyer and J Strother Moore, "MJRTY—a fast majority vote algorithm", in *Automated Reasoning: Essays in Honor of Woody Bledsoe*, Robert S. Boyer, Ed., vol. 1 of *Automated Reasoning Series*, pp. 105–117. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1991.

[44] Patrick Lincoln and John Rushby, "Formal verification of an algorithm for interactive consistency under a hybrid fault model", Tech. Rep. SRI-CSL-93-2, Computer Science Laboratory, SRI International, Menlo Park, CA, Mar. 1993, Also available as NASA Contractor Report 4527, July 1993.

[45] Albert L. Hopkins, Jr., Jaynarayan H. Lala, and T. Basil Smith III, "The evolution of fault tolerant computing at the Charles Stark Draper Laboratory, 1955–85", in *The Evolution of Fault-Tolerant Computing*, A. Avižienis, H. Kopetz, and J. C. Laprie, Eds., vol. 1 of *Dependable Computing and Fault-Tolerant Systems*, pp. 121–140. Springer-Verlag, Vienna, Austria, 1987.

[46] Jaynarayan H. Lala, "A Byzantine resilient fault tolerant computer for nuclear power application", in *Fault Tolerant Computing Symposium 16*, Vienna, Austria, July 1986, IEEE Computer Society, pp. 338–343.

[47] Patrick Lincoln and John Rushby, "Formal verification of an interactive consistency algorithm for the Draper FTP architecture under a hybrid fault model", in *COMPASS '94 (Proceedings of the Ninth Annual Conference on Computer Assurance)*, Gaithersburg, MD, June 1994, IEEE Washington Section, pp. 107–120.

[48] Ben L. Di Vito, Ricky W. Butler, and James L. Caldwell, "High level design proof of a reliable computing platform", In Meyer and Schlichting [86], pp. 279–306.

[49] John Rushby, "A fault-masking and transient-recovery model for digital flight-control systems", in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Jan Vytopil, Ed., Kluwer International Series in Engineering and Computer Science, chapter 5, pp. 109–136. Kluwer, Boston, Dordecht, London, 1993, An earlier version appeared in [85, pp. 237–257].

[50] Ricky W. Butler, Ben L. Di Vito, and C. Michael Holloway, "Formal design and verification of a reliable computing platform for real-time control: Phase 3 results", NASA Technical Memorandum 109140, NASA Langley Research Center, Hampton, VA, Aug. 1994.

[51] Ben L. Di Vito and Ricky W. Butler, "Formal techniques for synchronized fault-tolerant systems", in *Dependable Computing for Critical Applications—3*, C. E. Landwehr, B. Randell, and L. Simoncini, Eds. Sept. 1992, vol. 8 of *Dependable Computing and Fault-Tolerant Systems*, pp. 163–188, Springer-Verlag, Vienna, Austria.

[52] Steven P. Miller and Mandayam 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*, Boca Raton, FL, 1995, IEEE Computer Society, To appear.

[53] David W. Best, Charles E. Kress, Nick M. Mykris, Jeffrey D. Russell, and William J. Smith, "An advanced-architecture CMOS/SOS microprocessor", *IEEE Micro*, vol. 2, no. 4, pp. 11–26, Aug. 1982.

[54] W. C. Carter, W. H. Joyner, Jr., and D. Brand, "Microprogram verification considered necessary", in *National Computer Conference*. AFIPS Conference Proceedings, 1978, vol. 48, pp. 657–664.

[55] Jeffrey V. Cook, "Verification of the C/30 microcode using the State Delta Verification System (SDVS)", in *Proceedings of the 13th National Computer Security Conference*, Washington, DC, Oct. 1990, National Institute of Standards and Technology/National Computer Security Center, pp. 20–31.

[56] David May, Geoff Barrett, and David Shepherd, "Designing chips that work", In Hoare and Gordon [87], pp. 3–19.

[57] Warren A. Hunt, Jr., *FM8501: A Verified Microprocessor*, vol. 795 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, Berlin, 1994.

[58] Warren A. Hunt, Jr. and Bishop C. Brock, "A formal HDL and its use in the FM9001 verification", In Hoare and Gordon [87], pp. 35–47.

[59] Tom Gilb and Dorothy Graham, *Software Inspections*, Addison-Wesley, 1993.

[60] John Rushby, "Formal methods and digital systems validation for airborne systems", Tech. Rep. SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 1993, Also available as NASA Contractor Report 4551, December 1993.

[61] Robyn R. Lutz, "Analyzing software requirements errors in safety-critical embedded systems", in *IEEE International Symposium on Requirements Engineering*, San Diego, CA, Jan. 1993, pp. 126–133.

[62] John Guttag and J. J. Horning, "Formal specification as a design tool", in *7th ACM Symposium on Principles of Programming Languages*, Las Vegas, NV, Jan. 1980, pp. 251–261.

[63] N. Shankar, "Abstract datatypes in PVS", Tech. Rep. SRI-CSL-93-9, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 1993.

[64] J. H. Cheng and C. B. Jones, "On the usability of logics which handle partial functions", in *Proceedings of the Third Refinement Workshop*, Carroll Morgan and J. C. P. Woodcock, Eds. 1990, pp. 51–69, Springer-Verlag Workshops in Computing.

[65] Joseph R. Shoenfield, *Mathematical Logic*, Addison-Wesley, Reading, MA, 1967.

[66] John K. Ousterhout, *Tcl and the TK Toolkit*, Professional Computing Series. Addison-Wesley, 1994.

[67] Imre Lakatos, *Proofs and Refutations*, Cambridge University Press, Cambridge, England, 1976.

[68] Israel Kleiner, "Rigor and proof in mathematics: A historical perspective", *Mathematics Magazine*, vol. 64, no. 5, pp. 291–314, Dec. 1991, Published by the Mathematical Association of America.

[69] Robert E. Shostak, “On the SUP-INF method for proving Presburger formulas”, *Journal of the ACM*, vol. 24, no. 4, pp. 529–543, Oct. 1977.

[70] Robert E. Shostak, “Deciding linear inequalities by computing loop residues”, *Journal of the ACM*, vol. 28, no. 4, pp. 769–779, Oct. 1981.

[71] R. S. Boyer and J S. Moore, “Integrating decision procedures into heuristic theorem provers: A case study with linear arithmetic”, in *Machine Intelligence*, vol. 11. Oxford University Press, 1986.

[72] G. L. J. M. Janssen, *ROBDD Software*, Department of Electrical Engineering, Eindhoven University of Technology, October 1993.

[73] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas, “Effective theorem proving for hardware verification”, in *Preliminary Proceedings of the Second Conference on Theorem Provers in Circuit Design*, Ramayya Kumar and Thomas Kropf, Eds., Bad Herrenalb (Blackforest), Germany, Sept. 1994, pp. 287–305, Forschungszentrum Informatik an der Universität Karlsruhe, FZI Publication 4/94.

[74] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith, *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[75] F. Keith Hanna, Neil Daeche, and Mark Longley, “Specification and verification using dependent types”, *IEEE Transactions on Software Engineering*, vol. 16, no. 9, pp. 949–964, Sept. 1989.

[76] T. Yuasa and R. Nakajima, “IOTA: A modular programming system”, *IEEE Transactions on Software Engineering*, vol. SE-11, no. 2, pp. 179–187, Feb. 1985.

[77] Ralph Melton and David L. Dill, *Murϕ Annotated Reference Manual*, Computer Science Department, Stanford University, Stanford, CA, Mar. 1993.

[78] Kenneth L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, Boston, MA, 1993.

[79] J. R. Burch and D. L. Dill, “Automatic verification of pipelined microprocessor control”, in *Computer-Aided Verification, CAV '94*, David Dill, Ed., Stanford, CA, June 1994, vol. 818 of *Lecture Notes in Computer Science*, pp. 68–80, Springer-Verlag.

[80] D. Cyrluk and P. Narendran, “Ground temporal logic—a logic for hardware verification”, in *Computer-Aided Verification, CAV '94*, David Dill, Ed., Stanford, CA, June 1994, vol. 818 of *Lecture Notes in Computer Science*, pp. 247–259, Springer-Verlag.

[81] K. L. Heninger, “Specifying software requirements for complex systems: New techniques and their application”, *IEEE Transactions on Software Engineering*, vol. SE-6, no. 1, pp. 2–13, Jan. 1980.

[82] John Rushby and Mandayam Srivas, “Using PVS to prove some theorems of David Parnas”, in *Higher Order Logic Theorem Proving and its Applications (6th International Workshop, HUG '93)*, Jeffrey J. Joyce and Carl-Johan H. Seger, Eds., Vancouver, Canada, Aug. 1993, number 780 in Lecture Notes in Computer Science, pp. 163–173, Springer-Verlag.

[83] Costas Courcoubetis, Ed., *Computer-Aided Verification, CAV '93*, vol. 697 of *Lecture Notes in Computer Science*, Elounda, Greece, June/July 1993. Springer-Verlag.

[84] H. Langmaack, W.-P. de Roever, and J. Vytopil, Eds., *Formal Techniques in Real-Time and Fault-Tolerant Systems*, vol. 863 of *Lecture Notes in Computer Science*, Lübeck, Germany, Sept. 1994. Springer-Verlag.

[85] J. Vytopil, Ed., *Formal Techniques in Real-Time and Fault-Tolerant Systems*, vol. 571 of *Lecture Notes in Computer Science*, Nijmegen, The Netherlands, Jan. 1992. Springer-Verlag.

[86] J. F. Meyer and R. D. Schlichting, Eds., *Dependable Computing for Critical Applications—2*, vol. 6 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag, Vienna, Austria, Feb. 1991.

[87] C. A. R. Hoare and M. J. C. Gordon, Eds., *Mechanized Reasoning and Hardware Design*, Prentice Hall International Series in Computer Science, Hemel Hempstead, UK, 1992.

Sam Owre received his BS in mathematics from Stevens Institute of Technology in 1975, and an MA in mathematics from the University of California, Los Angeles in 1978. He is a Senior Software Engineer in the Computer Science Laboratory at SRI International, where for the past 5 years he has devoted most of his waking hours to the development of the PVS and EHDM verification systems. Prior to that he worked in a number of AI-related research projects at Advanced Decision Systems, and before that he built yet another verification system (described in the February 1987 issue of this journal) while working at Sytek Inc. He has coauthored a number of papers on formal methods.

He is a member of the IEEE, the Association for Computing Machinery, the Association for Symbolic Logic, the European Association for Theoretical Computer Science, and the American Mathematical Society.

John Rushby received BSc and PhD degrees in computing science from the University of Newcastle upon Tyne in 1971 and 1977, respectively. He joined the Computer Science Laboratory of SRI International in 1983, and served as its director from 1986 to 1990; he currently manages its research program in formal methods and dependable systems. Prior to joining SRI, he held academic positions at the Universities of Manchester and Newcastle upon Tyne in England.

Dr. Rushby leads the projects developing and applying EHDM and PVS. His research interests center on the use of formal methods for problems in design and assurance for dependable systems. He is the author of the section on formal methods for the FAA Digital Systems Validation Handbook.

He is a member of the IEEE, the Association for Computing Machinery, the American Institute of Aeronautics and Astronautics, and the American Mathematical Society. He is an associate editor for the Communications of the ACM and recently joined the editorial board of the journal “Formal Aspects of Computing.”

Natarajan Shankar received a B.Tech. degree in electrical engineering from the Indian Institute of Technology, Madras in 1980, and a PhD in computer science from the University of Texas at Austin in 1986. He has been a computer scientist with the Computer Science Laboratory at SRI International since 1989. Prior to joining SRI, he was a research associate with the Stanford University Computer Science Department. His interests include formal methods, automated reasoning, metamathematics, and linear logic. He co-developed SRI's PVS specification language and verification system. His book *Metamathematics, Machines, and Gödel's Proof* was recently published by Cambridge University Press.

Dr. Shankar is a member of the Association for Computing Machinery, the Association for Symbolic Logic, the European Association for Theoretical Computer Science, and the IFIP Working Group 2.3 on programming methodology.

Friedrich von Henke received the diploma and Dr.rer.nat. degrees from the University of Bonn, Germany, in 1971 and 1973, respectively. He has been a Professor of Informatics (computer science) at the University of Ulm, Germany, since 1990. From 1984 to 1990 he was a program manager in the Computer Science Laboratory of SRI International; prior to joining SRI International, he held research appointments in the Artificial Intelligence and Computer Systems Laboratories of Stanford University and at the German Research Center for Mathematics and Data Processing (GMD).

Dr. von Henke has been active in the area of formal methods since 1973. At SRI, he led the project developing the EHDM system. His current research interests include formal and knowledge-based methods of software development and their applications; in this area he also continues the collaboration with the group at SRI.

He is a member of ACM, IEEE Computer Society, GI (German Informatics Society), and the European Association for Theoretical Computer Science.