[42] Mandayam Srivas and Mark Bickford. Verification of the FtCayuga 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).

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

[44] J. Vytopil, editor. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, Nijmegen, The Netherlands, January 1992. Springer-Verlag.

[45] J. Lundelius Welch and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, April 1988.

[46] 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*, 66(10):1240–1255, October 1978.

[47] 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, April 1992. (Work performed by Computational Logic Incorporated).

[48] T. Yuasa and R. Nakajima. IOTA: A modular programming system. *IEEE Transactions on Software Engineering*, SE-11(2):179–187, February 1985.

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

[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] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

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

[33] John Rushby. A fault-masking and transient-recovery model for digital flight-control systems. In Jan Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Kluwer International Series in Engineering and Computer Science, chapter 5, pages 109–136. Kluwer, Boston, Dordecht, London, 1993. An earlier version appeared in [44, pp. 237–257].

[34] John Rushby. *Formal Methods and Digital Systems Validation for Airborne Systems.* Federal Aviation Administration Technical Center, Atlantic City, NJ, 1993. Forthcoming chapter for "Digital Systems Validation Handbook," DOT/FAA/CT-88/10.

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

[36] John Rushby and Friedrich von Henke. Formal verification of algorithms for critical systems. *IEEE Transactions on Software Engineering*, 19(1):13–23, January 1993.

[37] John Rushby, Friedrich von Henke, and Sam Owre. An introduction to formal specification and verification using EHDM. Technical Report SRI-CSL-91-2, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1991.

[38] Fred B. Schneider. Understanding protocols for Byzantine clock synchronization. Technical Report 87-859, Department of Computer Science, Cornell University, Ithaca, NY, August 1987.

[39] Natarajan Shankar. Mechanical verification of a generalized protocol for Byzantine fault-tolerant clock synchronization. In Vytopil [44], pages 217–236.

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

[41] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[15] F. Keith Hanna, Neil Daeche, and Mark Longley. Specification and verification using dependent types. *IEEE Transactions on Software Engineering*, 16(9):949–964, September 1989.

[16] R. M. Kieckhafer, C. J. Walter, A. M. Finn, and P. M. Thambidurai. The MAFT architecture for distributed fault tolerance. *IEEE Transactions on Computers*, 37(4):398–405, April 1988.

[17] Israel Kleiner. Rigor and proof in mathematics: A historical perspective. *Mathematics Magazine*, 64(5):291–314, December 1991. Published by the Mathematical Association of America.

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

[19] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[20] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

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

[22] 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, December 1993.

[23] Robyn R. Lutz. Analyzing software requirements errors in safety-critical embedded systems. In *IEEE International Symposium on Requirements Engineering*, pages 126–133, San Diego, CA, January 1993.

[24] 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.

[25] P. M. Melliar-Smith and R. L. Schwartz. Formal specification and verification of SIFT: A fault-tolerant flight control system. *IEEE Transactions on Computers*, C-31(7):616–630, July 1982.

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

[27] J. F. Meyer and R. D. Schlichting, editors. *Dependable Computing for Critical Applications—2*, volume 6 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag, Vienna, Austria, February 1991.

[28] Paul S. Miner. A verified design of a fault-tolerant clock synchronization circuit: Preliminary investigations. NASA Technical Memorandum 107568, NASA Langley Research Center, Hampton, VA, March 1992.

References

[1] W. R. Bevier and W. D. Young. Machine-checked proofs of a Byzantine agreement algorithm. Technical Report 55, Computational Logic Incorporated, Austin, TX, June 1990.

[2] W. R. Bevier and W. D. Young. The design and proof of correctness of a fault-tolerant circuit. In Meyer and Schlichting [27], pages 243–260.

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

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

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

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

[7] 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.

[8] 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*, pages 11–20, Virginia Beach, VA, October 1990. The Institute of Electrical and Electronics Engineers.

[9] Ben L. Di Vito and Ricky W. Butler. Formal techniques for synchronized fault-tolerant systems. In C. E. Landwehr, B. Randell, and L. Simoncini, editors, *Dependable Computing for Critical Applications—3*, volume 8 of *Dependable Computing and Fault-Tolerant Systems*, pages 163–188. Springer-Verlag, Vienna, Austria, September 1992.

[10] Ben L. Di Vito, Ricky W. Butler, and James L. Caldwell. High level design proof of a reliable computing platform. In Meyer and Schlichting [27], pages 279–306.

[11] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: An interactive mathematical proof system. *Journal of Automated Reasoning*, 11(2):213–248, October 1993.

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

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

[14] John Guttag and J. J. Horning. Formal specification as a design tool. In *7th ACM Symposium on Principles of Programming Languages*, pages 251–261, Las Vegas, NV, January 1980.

away at NASA Langley Research Center. The evolution of our languages and tools in response to the lessons learned 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 early detection of error, and the productive development of proofs whose information content can assist in the certification of safety-critical systems [34].

Most of the techniques we employ were pioneered by others. For example, Nuprl [7] and Veritas [15] provide predicate subtypes and dependent types; theory interpretations were used in Iota [48] and, later, Imps [11]; our theorem proving techniques draw on LCF [13], the Boyer-Moore prover [3, 5], and on earlier work at SRI [41]. 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, which 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 their mechanizations, 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.

Although this paper has concentrated on our experiences with verification of fault-tolerance properties, EHDM and PVS are also being applied to designs for secure systems, to hardware, and to real-time applications. In other collaborative projects, PVS is being used in requirements analysis for the "Jet Select" function of the Space Shuttle flight control system, and for microprogram verification of a commercial avionics computer.

Acknowledgments:

The work reported here owes a very great deal to our collaborators at NASA Langley Research Center: Rick Butler, Jim Caldwell, Paul Miner, and Ben Di Vito. We also thank colleagues at SRI: Pat Lincoln and Erwin Liu, who performed some of the verifications, and David Cyrluk, who contributed to the tools development.

as a lambda-application, but must print as a "let," not a "lambda," and this must remain so 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.

Finally, we have found it useful to provide some automated support for instantiation of quantified variables and for conducting proofs by induction. These capabilities are visibly impressive, and nice to have, but their impact on productivity is relatively minor.

4 Conclusions

We have described our experiences in developing mechanically-checked formal verifications of several quite difficult arguments arising in fault-tolerant systems. As well as ourselves, verifications were performed by colleagues at SRI who had not been involved in the development of our tools, and by collaborators 3,000 miles general principles. For example, we may extract and generalize some part of the specification as a reusable and verified component to be stored in a library.

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.

We have made a number of design decisions in the interests of enhancing productivity for the human user 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.

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.

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 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 prover is like this), or in the form of a program that specifies the proof strategy to be used (the "tactics" of LCF-style provers [13] such as HOL 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 if the user can specify customized proof control "strategies" (similar to LCF-style tactics and tacticals) that can automate the repetitive elements of the proof.

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 (for example, resolution) work in ways that do not lend themselves to the extraction of a readable proof, and are unattractive on this count. On the other hand, heuristic methods can generate "unnatural" proofs, while low-level proof checkers overwhelm the reader with 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 major steps, while routine ones are heavily automated.

Generalization and Maintenance: Designs are seldom static; user requirements may change with time, as may the interfaces and services provided by other components of the overall system. A verification may therefore need to be revisited periodically and adjusted in the light of changes, or explored in order to predict the consequences of proposed 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 to the specification of lemmas will not derail it.

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, or to distill

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

$$x \leq y \ \wedge \ x \leq 1 - y \ \wedge \ 2 \times x \geq 1 \ \supset \ F(2 \times x) = F(1)$$

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

Our proof checkers include decision procedures for an extended, quantifier-free form of Presburger Arithmetic: that is arithmetic with constants and variables, addition and subtraction, but with multiplication restricted to the linear case (i.e., multiplication by literal constants only), together with the relations $<$, $>$, $\leq$, $\geq$, $=$, and $\neq$. The decision procedures deal with both integer and rational numbers, and propositional calculus [41]. 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 operations in proofs arising from formal verification are to expand the definition of a function and to replace one side of an equation by the corresponding instance of the other. Both of these can be automated by rewriting. But 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 [4]. More visibly impressive capabilities such as automatic induction heuristics 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 in the order and selection of results they are invited to consider (the Boyer-Moore replay a proof, this is done only on request. "Proof-tree analysis" (described below) identifies the state of a proof during an evolving verification.

3.3 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 [18]. 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 [17].

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.

(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. Not all verification systems police these matters carefully. For example, some implementations of HOL, which is often praised as a system with very sound foundations, can still consider a theorem proved after some of its supporting definitions have been changed.

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 example, in demonstrating the consistency of the axiomatization used to specify assumptions about clocks [36], 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.

3.2 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 direct use of mathematical symbols such as $\forall$. 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. Since the marketplace has plainly demonstrated that the preferred and most productive connection between mind and computer is the Emacs editor, we have adopted this as our interface, and accepted an ascii representation for our specifications. Our experience has been that it is the naturalness of its semantic foundation and syntactic expression that determines the acceptance of a specification notation, not its lexical representation. 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.

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.

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 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.[12] Our mechanization is based on the notion of *theory interpretations* [40, 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.[13]

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.[14] 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

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

[13]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.

[14]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.

automatically generate the proof obligations necessary to ensure that the operations preserve the invariant.

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 [6] 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 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.

$$\begin{array}{l}
\mathit{subp}((i\ :\ \mathit{int}),\ (j\ :\ \mathit{int} \mid i \geq j)){:}\ \textbf{recursive}\ \mathit{int} = \\
\quad \textbf{if}\ \ i=j\ \ \textbf{then}\ \ 0\ \ \textbf{else}\ \ \mathit{subp}(i,\ j+1)+1\ \ \textbf{endif} \\
\ \ \textbf{measure}\ \lambda\ (i\ :\ \mathit{int}),\ (j:\ \mathit{int} \mid i \geq\ j):\ i-j
\end{array}$$[10]

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

$$[i : \mathit{int}, \{j : \mathit{int} \mid i \geq j\}]$$

(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.

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.[11] 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")

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

[11]Abstract data types are also conveniently specified axiomatically. PVS provides for this through a "datatype" construction that generalizes the "shell" mechanism of the Boyer-Moore prover [3]. PVS datatype specifications automatically generate axiomatizations of a stereotyped form that is known to be consistent.

is immediately seen to be type-correct. However, the expression

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

is not obviously type-correct (since the outermost *pop* requires a *nonempty_stack*, but is given the result of another *pop*—which is only known to be a *stack*). However, it can be shown to be well-typed by proving the theorem

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

which follows from the usual stack axioms. EHDM and PVS automatically generate this theorem as a proof obligation (i.e., TCC) when typechecking the expression (1).

Proof obligations 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 proof obligations 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:

$$injection : \mathbf{type} = \{f : [t_1 \rightarrow t_2] \mid \forall (i, j : t_1) \colon f(i) = f(j) \supset i = j\}$$

(here t_1 and t_2 are uninterpreted types introduced in the module parameter list). If we were later to specify the function *square* as an injection from the integers to the naturals by the declaration

$$square : injection[int \rightarrow nat] = \lambda(x : int) \colon x \times x : nat$$

then the PVS typechecker would require us to show that the body of *square* satisfies the *injection* subtype predicate.[9] That is, it requires the proof obligation $i^2 = j^2 \supset i = j$ to be proved in order to establish that the *square* function is well-typed. Since this theorem 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. Yet another example of the utility of predicate subtypes 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

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

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

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

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 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 appears as an antecedent in the proof obligation. These proof obligations establish that the value of the original expression does not depend upon the value of a type-incorrect term. The arithmetic decision 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 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 the stack operations as follows:

stack: **type**
empty: *stack*
nonempty_stack: **type** = {(*s: stack*) | *s* ≠ *empty*}

push: [*elem*, *stack* → *nonempty_stack*]
pop: [*nonempty_stack* → *stack*]
top: [*nonempty_stack* → *elem*]

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. For the same reason, the construction

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

We do use specialized formalisms, such as temporal logic, when it seems appropriate, but we do so by formalizing them within higher-order logic. The advantage of embedding such formalisms within a single logic is that it is then easier to combine them with others 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 fairly closely [19,38], 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 [10] with a fully formal version [33]).

We have taken some pains to allow these formal specifications to be rendered in a natural syntactic form. For example, we provide rich sets of propositional connectives (including a polymorphic *if-then-else*) and of arithmetic and relational operators, and we allow set-notation for predicates.[7] 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, boolean, and integer and rational numbers) and (higher-order) function types. We soon found it convenient to add record and enumeration types, and then—the most significant step of all—predicate subtypes. In PVS we also provide tuple types, and dependent type constructions.[8]

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:

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

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 [14].

[7]In higher-order logic, sets are represented by their characteristic predicates, which are themselves simply functions with range type "boolean."

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

soundness of axiomatizations are clearly desirable (purely definitional specifications are often too restricting), as are habits and 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.

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.

3.1 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, simply-typed higher-order logic is adequate for formalizing all 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.[6]

[6] EHDM does support Hoare logic directly, and we have used this capability in other applications. Even so, we generally find it most convenient to develop the bulk of the specification and verification

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 most ambitious formal verification carried out in the program so far was performed by Rick Butler and Ben Di Vito at NASA: it connects fault masking with clock synchronization. The models for fault masking (and also those for interactive consistency) assume totally synchronous execution of the redundant computing channels (and instantaneous communication), whereas the clock-synchronization algorithms guarantee only that the channels are synchronized within some small bound. The reconciliation of these different models involves a hierarchical verification with two intermediate levels. 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 fault-masking model, now called the replicated synchronous (RS) model; below this is the distributed synchronous (DS) model, which introduces the fact that communication between channels takes time; and at the bottom is the distributed asynchronous (DA) model, which connects to the clock synchronization conditions and recognizes that the channels are only approximately synchronized. The US to RS verification is similar to ours, the other two are new and involve quite large specifications and proofs (well over 300 lemmas) [9].

It will be noted that we have concentrated on verifying algorithms and architectural designs (rather than program code or hardware circuits); the available evidence points to these and other early lifecycle concerns (particularly requirements) as the principle sources of failure in safety critical systems.[5]

3 Lessons Learned

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

[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 [23] reports on 387 software faults detected during integration and system testing of the Voyager and Galileo spacecraft. 197 of the faults were characterized as having potentially significant or catastrophic effects (with respect to the spacecraft's missions). Only 3 of these faults were programming errors; the vast majority were requirements problems.

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 [32].

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 (starting from the specification and a couple of minor lemmas) in under an hour. Thus equipped, we turned to an important variation on the algorithm due to Thambidurai and Park [43] 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 our colleague Pat Lincoln formally verified its correctness—and then found that it, too, was flawed.

How could we verify an incorrect algorithm? The explanation is that our axiomatization of a certain "hybrid majority" function required "omniscience": it had to be able to exclude faulty values from the vote, whereas the source of all the difficulty in Byzantine fault-tolerant algorithms is that it is not known which values are faulty. Thus our algorithm was "correct" but unimplementable. Pat Lincoln detected this problem by thinking carefully about the specification, and we were then able to develop and formally verify a new and correct algorithm for Interactive Consistency under a hybrid fault model [21]. This work took less than two weeks, and was primarily undertaken by Pat Lincoln (using PVS) as his first exercise in mechanized formal verification. We believe that the discipline of formalism and the mechanical support provided by PVS were instrumental in developing a correct algorithm and argument for this tricky problem.

A model for the main fault-masking and transient-recovery 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 [10]. Working in parallel, we developed a formal specification and verification of a slightly simplified, but also rather more general model [33]. 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 proves 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 was his first exposure to formal hardware verification). During circuit design, it became apparent that one of the assumptions of the clock-synchronization verification (i.e., 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.

We are now contemplating further adjustments to the verification. Dan Palumbo and Lynn Graham of NASA built equipment for experimenting with clock-synchronization circuitry and found that the observed worst-case skews were better than predicted by theory. They showed that a slight adjustment to the analysis can bring theory into closer agreement with observation [30]. We intend to incorporate this improved bound into our mechanically-checked verification, and will also expand the analysis to incorporate a hybrid fault model (an informal derivation has already been developed).

There are alternatives to ICA that seem more attractive from the implementation point of view. 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 appears 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 [38], 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 [39].

This verification included a proof that the "convergence function" of ICA satisfies Schneider's general conditions (thereby providing an independent formal verification of ICA). Paul Miner of NASA took a copy of our EHDM verification and extended it to verify that the more attractive convergence function characterizing the Welch-Lynch fault-tolerant mid-point algorithm [45] also satisfies these conditions. In addition, he identified improvements in the formulations of some of the conditions. In continuing work, he is verifying a significant extension to the algorithm that provides for transient recovery [28].

Turning from fault-tolerant clock synchronization to sensor distribution, we next focussed on the "Oral Messages" algorithm for Interactive Consistency [20].[4] Bevier and Young at CLI, who had already verified this algorithm, found it "a fairly difficult exercise in mechanical theorem proving" [1]. We suspected that their treatment

[4] Interactive consistency is the problem of distributing consistent values to multiple channels in the presence of faults [31]. 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.

program “state” is supported in EHDM, it is not used in the specifications considered here; algorithms and computations are described functionally. 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). The type systems of both languages provide features (such as predicate subtypes) that render typechecking algorithmically undecidable. In these cases, proof obligations (called type-correctness conditions, or TCCs) are generated and must be discharged before the specification is considered type correct. The specification languages of EHDM and PVS are built on very different foundations than those of, say, Z and VDM, but, in our experience, provide similar convenience and expressiveness.

2 Formal Verifications Performed

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 undertaken and we began by simply trying to reproduce the arguments in the journal paper that introduced the algorithm [19]. Eventually, we succeeded, but discovered in the process that the proofs or statements of all but one of the lemmas, and 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 [35, 36]. Our mechanized verification in EHDM took us a couple of months to complete and required about 200 lemmas (most 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. Later, when definitional forms guaranteeing conservative extension were added to EHDM, we were able to eliminate the large majority of these in favor of definitions. Even so, Bill Young of CLI, who repeated our verification using the Boyer-Moore prover [47], pointed out 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. We have since verified the consistency of the axioms in our current specification of ICA using the theory interpretation mechanism of EHDM.

Our colleague Erwin Liu developed a design for a hardware circuit to perform part of the clock-synchronization function, and formally verified the design [22] (this us for some time to come.[3] The verifications performed with our tools are described in Section 2, the lessons we have learned in Section 3, and brief conclusions are presented in Section 4. Before describing the verifications performed with them, we briefly introduce our tools.

1.1 Our Verification Systems

EHDM, which first became operational in 1984 [26] 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. EHDM's specification language is a higher-order logic with a rather rich type system and facilities for grouping related material into parameterized modules. EHDM supports hierarchical verification, so that 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. The EHDM tools include 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 [41]) 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 [37].

Development of PVS, our other verification system, started in 1991; it was built as a lightweight prototype for a "next generation" verification system 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 logic of PVS is similar to that of EHDM, but has an even richer type system. Its theorem prover includes the same decision procedures as EHDM, but in an interactive environment that uses a presentation based on the sequent calculus [29]. 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. Theorem proving techniques prototyped in PVS are now being incorporated into EHDM.

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. Although a notion of implicit

[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 [2] and [42], respectively. The overall program is not large; it is equivalent to about three full-time staff at NASA, and slightly less than one each at CLI, ORA, and SRI.

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 same is true of the clock synchronization that keeps the channels operating in lock-step. Byzantine fault-tolerant algorithms suitable for both problems are known, but suffer from some disadvantages. First, the classic 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, 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 simple majority voting, and as many as four crash faults. Yet a standard 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 various types [43].

Although the principles just sketched for a "reliable computing platform" (RCP) for flight-control applications are 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 an RCP. 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 nonreconfigurable RCP with transient recovery.

This is a rather ambitious goal, since the arguments for correctness of some of the individual Byzantine 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

[2]Some aspects of SIFT—which was built for NASA Langley—were subjected to formal verification [25], but the treatment was far from complete.

1 Introduction

Catastrophic failure of digital flight-control systems for passenger airplanes must be "extremely improbable"; a requirement that can be interpreted as a failure rate of less than 10^{-9} per hour [12, 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 this. 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 mechanisms for ultra-high reliability is a challenging problem. Redundancy management can account for half the software in a flight-control system [8] and, if less than perfect, can itself become the *primary* source of system failure [24].

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 [46]: several independent computing channels 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] [31], so that each good channel gets exactly the same input data; the channels run the same application tasks on the same data at 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 output. The original SIFT design suffered from performance problems, but several effective architectures based on this general idea have since been developed, including one (called MAFT) by a manufacturer of flight-control systems [16] that improved on many of the details.

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 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 *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

[1]Byzantine faults are those that manifest 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. Crash faults are as if the failed channel had simply ceased to exist.

Formal Verification for Fault-Tolerant Architectures: Some Lessons Learned[*]

Reprint (slightly expanded) of a paper in *FME '93: Industrial-Strength Formal Methods*, pages 482–500, Odense, Denmark, April 1993. Volume 670 of Springer Verlag *Lecture Notes in Computer Science*. **Note: A major revision of this paper appears in IEEE Transactions on Software Engineering, Vol. 21 No. 2 (feb. 1995) pp. 107–125 under the title "*Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS*."**

Sam Owre, John Rushby,
Natarajan Shankar, Friedrich von Henke[†]

Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

Abstract

In collaboration with NASA's Langley Research Center, we are developing mechanically verified formal specifications for the fault-tolerant architecture, algorithms, and implementations of a "reliable computing platform" (RCP) for life-critical digital flight-control applications. To achieve a failure rate that is certifiably less than 10^{-9} per hour, RCP employs redundant, synchronized computing channels for fault masking and transient recovery. The synchronization and sensor-distribution algorithms are resilient with respect to a hybrid fault model that includes Byzantine faults.

Several of the formal specifications and verifications performed in support of RCP are individually of considerable complexity and difficulty. But in order to contribute to the larger 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, build on, modify, or cannibalize an intricate verification.

Accordingly, we have been developing and honing our verification tools to better support these large, difficult, iterative, and collaborative verifications. Our goal is to reduce formal verifications as difficult as these to routine exercises, and to maximize the value obtained from formalization and verification.

In this paper, we describe some of the challenges we have faced, lessons learned, design decisions taken, and results obtained.

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

[†]Von Henke's main affiliation is now Fakultät für Informatik, Universität Ulm, Germany.