Design Choices in Specification Languages and Verification Systems

Reprint of an invited paper from: Phillip Windley, editor, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*, Davis, CA, Aug. 1991, pages 195–204; IEEE Computer Society.

John Rushby

Computer Science Laboratory
SRI International
Menlo Park, CA 94025 USA

Abstract

We describe some of the design choices that should be considered in the development and application of specification languages and verification systems. A principal issue is the need to reconcile the desire for expressiveness in the specification language with the ability to provide effective mechanical support. We argue that this reconciliation is assisted by a novel approach to specification language design that requires theorem proving to be used during typechecking.

A second key requirement is for the theorem prover to be specialized towards the needs of verification. This means that the theorem prover must assist in the rapid identification of the sources of errors in incorrect theorems as well as in the certification of true theorems, and that it must produce a proof suitable for human review. We argue that a combination of powerful automation (including decision procedures) at the low level, and user-guidance at the high level, provides the most effective approach to these goals.

1 Introduction

SRI International's Computer Science Laboratory has been involved in formal specification and mechanically-assisted formal verification almost since the very beginnings of those fields. The 1970's saw the development of the Jovial Verification System [11], the Hierarchical Development Methodology (HDM) and its specification language Special [20, 21, 32], and Shostak's decision procedures for arithmetic [28–30]. In addition, Boyer and Moore were at SRI through most of the 1970's and much of the early version of their prover [5] was developed here.

During the 1980's our activity in formal verification was centered around EHDM [25].[1] Development of EHDM began in 1983, about the same time as HOL and, like HOL, it is based on higher-order logic. We have improved and enhanced EHDM over the years, and the current version is a mature, carefully engineered system that we and others have used to perform some very hard verifications of fault tolerant algorithms [23,24,26], verifications of hardware circuits [8], and numerous other examples. More recently, we have been thinking about a "next-generation" verification system for the 1990's and have constructed a research prototype, provisionally known as PVS, to probe some of the issues that we consider important.

Coming from outside the HOL community, but with a shared interest in formal specification and verification using higher-order logic, the design choices that we have explored may be of interest to those contemplating the further development of HOL itself.

2 What is Formal Verification For?

The most basic decision, and one that drives almost all other design choices, is to identify the purpose for which formal verification is to be performed and which the formal verification system is intended to support. There are several plausible purposes, of which I will consider two, "ultra-high assurance" and "contributing to engineering practice," in more detail.

2.1 Ultra-High Assurance

There are two elements to the production of high-quality systems: *quality control* is concerned with methods for eliminating defects, and *quality assurance* is concerned with methods for demonstrating the absence of defects. Those who take "ultra-high assurance" as the goal of formal verification value its contribution to the second of these; they consider that formal verification provides the most searching and complete scrutiny of a system design and that it provides the best assurance for the absence of design defects. From this perspective, the highest priority for the conduct of formal verification, and for the design and implementation of tools to support it, must be to limit the sources of potential error within the verification process itself.

The sources of potential error in formal methods are no different to those in any other application of applied mathematics in engineering [1]. An aerodynamic calculation that produces the result "$\mathit{lift} = 50.73$" may be considered a proof of the theorem that the design satisfies the requirement "$\mathit{lift} \geq 50$," and the utility of the calculation depends on the fidelity of the mathematical models underlying aerodynamics to the real behavior of air over a wing, on the veracity of the stated requirement, and on the extent to which the calculation is performed without error. Formal verification is subject to the same three limitations on its utility: the model of real-world behavior may be flawed (e.g., some assumption about a device manipulated by the program may be incorrect, or the system may be coded in a programming language whose implementation does not coincide with its assumed semantics); the requirements on the system may not be fully understood, or may be stated incorrectly in its specification; and the proof that the specification of the system design satisfies the specification of its requirements may be flawed.

[1] The research prototype that preceded EHDM was called STP [27,31]. It was used in the partial verification of some properties of the SIFT fault-tolerant computer design [19,34].

One way to limit modeling errors in formal verification is to take the "top to bottom" approach. If the concern is that the compiler might supply different semantics than those assumed for the programming language concerned, then verify the compiler. If the hardware might behave differently than assumed, then verify its design down to, say, the register-transfer level (RTL). If the CAD system that converts RTL descriptions into hardware might be flawed, then verify the CAD system, or alternatively bypass the CAD system and develop and verify the design down to gates—or even transistors. The CLI "stack" is an example of this approach [3]: it comprises a verified compiler, assembler, and microprocessor design taken down to the gate level. (A variant "short stack" provides an operating system kernel [2]).

The second source of flaws in formal verification is perhaps the most difficult to circumscribe: the possibility that its formal specifications do not accurately capture the properties required of the system cannot be eliminated by purely formal means. However, constraints on the formal methods employed can eliminate some forms of errors. For example, unrestricted use of axioms can introduce inconsistencies and thereby render specifications meaningless. Consequently, some specification languages enforce a principle of definition that excludes this possibility. The disadvantage of this approach is that specifications may become overly prescriptive, suggesting *how* something is to be achieved, rather than simply *what* is required.

The desire to ensure that no flaw may creep into a mechanically-checked proof gives rise to the desire for *manifestly* secure methods of proof: that is, methods of proof that ultimately reduce to applications of elementary, and generally accepted, rules of inference such as those of natural deduction or the sequent calculus. HOL and other verification systems derived from LCF [13] accomplish this in a very elegant, and relatively efficient, manner.

2.2 Contributing to Engineering Practice

The approach just described values formal verification as a method for quality assurance: that is, as a method for certifying that a design is free of (certain) defects. The argument against this approach is partly philosophical, but largely economic. The philosophical argument observes that the main sources of potential error in formal verification are intrinsic to its character as a modeling activity. Since models of the real world and of user requirements are different from the realities they attempt to describe, complete fidelity between model and reality can never be obtained. At some point, human engineering judgment must decide that a certain formal model is "close enough," and other methods of assurance, including empirical validation, must be used to buttress that provided by formal means. It is then reasonable to question how resources should be apportioned between formal methods and other techniques for quality assurance and control. Those who value formal verification as a contribution to general software and hardware engineering practice take a pragmatic approach to this question and treat formal methods as just another tool at their disposal, to be used wherever they are most effective. The interest then is in finding elements of the engineering process where formal methods can provide a significant improvement over standard practice.

Now, current practice in software and hardware engineering is rather well developed and, on the whole, rather effective. One of the key concepts in modern software engineering is the system life-cycle model. Its premise is that development and implementation are carried out in several distinguishable, sequential phases, each performing unique, well-defined tasks, and requiring different skills. One of the outputs of each phase is a document that serves as the basis for evaluating the outcome of the phase and forms a guideline for the subsequent phases. There have been many refinements to the basic model: Royce's *Waterfall* model [22], for example, recognized the existence of feedback between phases and recommended that such feedback should be confined to adjacent phases; Boehm's *spiral* model [4] advocates a more iterative approach, including rapid prototyping (see also Brooks [7]).

In all cases, the motivation is to try and catch errors as soon as possible. The more phases of the life-cycle that separate the commission and detection of an error, the more expensive it is to correct. It is usually cheap and simple to correct a coding bug caught during unit test, and it is usually equally simple and cheap to insert a missed requirement that is caught during system requirements review. But it will be ruinously expensive to correct such a missed requirement if it is not detected until the system has been coded and is undergoing integration test.

An important methodology in software engineering that endeavors to ensure that all errors are caught, and caught early, is called "Verification and Validation" (V&V). *Verification* is the process of determining whether each level of specification, and the final code itself, "fully and exclusively" implements the requirements of its superior specification. *Formal* verification is precisely a formalization of traditional verification, which is a manual examination of design documentation. *Validation* is the process by which delivered code is directly shown to satisfy the original user requirements by testing it in execution. Put another way, verification is concerned with demonstrating that the system is being built right, while validation is concerned that the right system is being built. The two processes are complementary: each is effective at detecting errors that the other will miss, and they are therefore usually employed together. Very similar combinations of techniques are employed in hardware design.

There is general agreement that V&V is very effective in the later stages of the lifecycle, and that the early stages are the most problematical. Brooks, for example observes [7]:

> "I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation. We still make syntax errors, to be sure, but they are fuzz compared with the conceptual errors in most systems."

A specific and well documented example of Brooks' observation is provided by the flight control system of the AFTI-F16 aircraft. Failures encountered during flight tests were largely due to defects in the basic architecture of the redundancy management system [18].

Anecdotal evidence[2] from hardware engineering leads to similar conclusions, suggesting that errors from the later stages of design and implementation are generally caught and eliminated before fabrication. However, some defects introduced in the early stages of the lifecycle apparently survive undetected and result in more than half of all VLSI designs being found defective at first fabrication.

Given these data and the desire to use formal methods as a cost-effective tool, we can identify a number of potential contributions. The general idea is to concentrate on those parts of the design process where existing methods are ineffective, expensive, or absent.

[2] I heard this from Kurt Keutzer in his keynote talk at this meeting.

- Formal specification may be used to replace, or supplement, documentation normally presented in natural language. Compared with informal documents, formal specifications are more precise and are free of ambiguity. Simple forms of mechanical analysis (such as parsing, typechecking, and cross-referencing) can detect certain types of error and facilitate tracking the consequences of changes.

- Formal methods can be used to gain a refined understanding of a class of designs, leading to a clarification of concepts, and to an orderly and uniform presentation. The result of this process may be generic and reusable specifications that can provide a common foundation for many similar projects. Examples of such contributions include development of a system architecture for oscilloscope design [10], and for systematic verification of microprocessor designs [35].

- The methods of formal verification may be used to perform "what if" analyses of requirements specifications. For example, the specification of a particular set of requirements can be evaluated by posing questions such as "if I've stated this right, then the following ought to be true, and the following ought to be impossible." These questions can be formulated as putative theorems about the specification, and formal verification techniques can then be used to probe their truth or falsity. This technique provides a form of validation that can be applied early in the design process, when regular testing is infeasible.

- Some parts of a system design may be subjected to formal verification, but for the purposes of finding errors and for gaining insight and understanding, rather than for certifying assurance. Since cost-effectiveness is important, this examination may be restricted to only the most critical, or difficult parts of the system; alternatively, the whole system may be examined at a high level of abstraction. Among the benefits that may be obtained from such an exercise is identification of the assumptions on which a verification rests. These assumptions correspond to requirements on other components of the system, and to properties of the physical world whose satisfaction must be established empirically, or by other non-formal means.

At SRI, we have chosen to concentrate on maximizing the contribution that formal methods can make to general engineering practice. We have done so because we perceive it to represent both a more urgent need and a more interesting engineering challenge than ultra-high assurance. A verification system that is to make a contribution to general engineering practice must provide a specification language that is accessible to the whole design team, not just to verification specialists, and that is not too far removed from the other notations (such as programming languages, and entity-relationship and dataflow diagrams) that will also be employed in a large project. At the same time, it must have a rigorous semantics and must support formal analysis. When theorem proving is employed as a means of exploration rather than certification, the efficiency of the process becomes critical, as does the quality of the information returned from both failed and successful proof attempts.

In the following sections, we discuss some of the design choices in specification languages and verification systems that can maximize their effectiveness in engineering practice.

3 Specification Languages

The purpose of specification is communication, and we believe that goal is best served by choosing a formalism as close as possible to that of conventional mathematical presentations. This discourages use of nonclassical logics and suggests a notation based on the full (i.e., quantified) first-order predicate calculus.

Not only is it important for a specification language to provide a familiar and reasonably rich logic, we consider it important that it should, to the extent possible, express that logic in familiar and attractive notation. Thus, functions should be able to take several arguments and function application should be written in the usual style: $f(x, y, z)$. Similarly, we should expect to be able to write arithmetic and certain other operators and relations in their familiar infix form.

A rich set of propositional connectives can enhance the readability of the specification. For example, a polymorphic *if-then-else* allows us to write expressions such as

$$x = \textbf{if } b \textbf{ then } y \textbf{ else } z.$$

This is much more readable than the equivalent formula

$$b \supset x = y \wedge \neg b \supset x = z$$

and there is no reason why the specification language should not silently perform this "if-lifting" transformation and allow the specifier to use the more convenient *if-then-else* form. It is convenient if the more general *case* expression is provided as well.

For specification, a multi-sorted (or typed[3]) system is generally preferred to an untyped system. There are two reasons for this. First, it is natural and convenient in computer science to distinguish the different types of entities that are manipulated: booleans from integers, lists from functions, and so on. By recognizing these distinctions in the specification language, we can encourage clear specifications and can detect many simple errors at an early stage using strict typechecking.[4] Second, efficient theorem proving requires that certain types of entities (for example booleans and integers) are recognized and treated specially; clearly this is done most naturally when the specification language supports the necessary type distinctions. In addition to the arithmetic and function types familiar in logic, a specification language should support type-constructions similar to those found in programming languages, such as records, tuples, and enumerations. More ambitiously, it should support the compact specification of abstract data types in a manner similar to, or more general than, the *shell* mechanism of the Boyer-Moore system [5].

A specification language having the characteristics described so far would be little more than a syntactically embellished version of predicate calculus. Experience shows that in computer science, as in mathematics, a more powerful formal system than predicate calculus is convenient, or is required, for many specification problems. Some of the most delicate choices in specification language design arise when the underlying formal system is extended beyond predicate calculus. Usually, a major design trade-off must be negotiated between

[3]In logic, sorts and types are different notions. However, we will follow computer-science terminology in which the distinction is blurred and the single term "type" is generally used.

[4]Not everyone favors the restrictions of a typed logic (see Lamport [17], for example). For the most part, the objections are to the excessively strict type-rules that are required for deterministic typechecking. Later on we will describe ways of relaxing these rules.

the convenience and expressiveness of the specification language, and the automation and effectiveness of the mechanical support that can be provided.

In this trade-off, the extremes are illustrated by the Z notation [33] from Oxford University, and the computational logic of Boyer and Moore [5, 6]. Z is very rich specification notation based on set theory which has been used to document several system designs [15], but for which only elementary mechanical support is available. The Boyer-Moore system, on the other hand, uses a very limited, raw logic as its specification language, but has a very powerful theorem prover that has been used to undertake several large and complete formal verifications [3]. Neither of these choices is ideal, in our opinion, and we believe there is a middle way that avoids the need to trade-off expressiveness against effective mechanical support. The key design choices that make this possible are the use of higher-order logic, and the use of theorem-proving during typechecking.

If we take the ability to specify sets soundly as a key requirement for an expressive specification language then, as in mathematical logic, we have two traditions to draw on. In axiomatic set theory, predicates come first, then sets (the antinomies are avoided by careful selection of the set-construction axioms), and functions are introduced as sets of pairs. In simple type-theory (i.e., higher-order logic), functions are primary, predicates are functions with range type boolean, and sets are identified with their characteristic predicates. Here, the antinomies are avoided by type-rules that prohibit the vicious-circle constructions [14].

The problems with axiomatic set theory as a foundation for a specification language are firstly that strict typechecking has to be grafted on[5]—and this is not straightforward—and secondly that functions are essentially partial. Partial functions further complicate strict typechecking, and are inimical to efficient theorem proving. In higher-order logic, on the other hand, strict typechecking is natural, and functions are (usually defined to be) total.

But if the choice of higher-order logic solves many problems, it does not solve all of them. We still have to decide what to do with constructions such as $\frac{x}{y}$ in the case that y is zero. One approach treats division as a total function and asserts that $\frac{x}{0}$ has some value, but provides no axioms that enable its value to be deduced. This simple approach becomes less so when expressions such as $\frac{x}{y} \times y$ are encountered, since the constraint on application of the cancellation law (i.e., $y \neq 0$) has to be checked (perhaps repeatedly) at proof-time. Alternative treatments, which allow true partial functions, require the complications of undefined terms and three-valued logics [9] or, more elegantly, logics of partial terms [12].

A quite different choice, which is the one we prefer, takes division to be a total function, but on the signature $\mathcal{Q} \times \mathcal{Q}_z \rightarrow \mathcal{Q}$ where $\mathcal{Q}$ denotes the rationals and $\mathcal{Q}_z$ denotes the nonzero rationals. Our specification languages allow subtypes to be associated with predicates. Thus $\mathcal{Q}_z$ is the subtype of the rationals associated with the predicate $(\lambda y . y \neq 0)$. Typechecking an expression of the form $\frac{x}{y}$ incurs the obligation to prove that y satisfies the defining predicate $y \neq 0$ in the context of its use.

To see how this works in practice, consider a simplified fragment from one of the constraints on a clock synchronization algorithm [24]: $\frac{\rho}{n-m}$, where ρ is a rational, n and m are naturals, and $n > m$. We have no subtraction operation defined on the naturals, so n and m are promoted to integers (a supertype of naturals), and $n - m$ is interpreted as integer subtraction, yielding an integer result. We are now supplying an integer as the second argument to the division operator, which requires a nonzero rational in this position. The rationals are a common supertype to both the integers and the nonzero rationals, so

[5] In raw set theory, constructions such as $x \cup \{x\}$ are perfectly acceptable.

$n - m$ can be promoted to a rational, and then reduced to the nonzero rational required for type-correctness—provided we can prove the theorem $n - m \neq 0$, which in this case follows obviously from the constraint $n > m$.

The implementations of our specification languages attempt to prove these type-correctness theorems automatically. Those that cannot be proved within some small time limit (typically one tenth of a second) are added to the set of proof obligations that must be discharged by user-directed proofs.

Our subtyping approach applies to function types as well. This allows us to define the injections, for example, as a subtype of the function types. If we declare an injection and then define it by construction, we will be required to prove that the construction does indeed satisfy the defining predicate for injections.

We have found the willingness to use theorem proving during typechecking to be very liberating. In programming languages it is natural that typechecking should be required to be deterministic, but this requirement is unduly constraining in the context of a specification environment equipped with a powerful theorem prover. Additional benefits that derive from the decision to use theorem proving in typechecking include a straightforward treatment of dependent and quotient types, and the ability to address the objections of those, such as Lamport [17], who find strict typechecking irksome. A user who knows that, for possibly deep reasons, the expression e should have type t can simply write $e : t$ and the system will generate the necessary type-correctness theorem for the user to prove. Thus, sophisticated type-determinations can be controlled by the user, without having to sacrifice the convenience of automation in simple cases.

The logical foundations of a specification language determine what can be expressed at all, and also what can be expressed conveniently. In addition, most specification languages encourage or mandate a certain *style* of specification. For example, some specification languages require constructive specifications, in which each new operation is specified in terms of a mechanism to achieve it. The advantage of such constructive specifications is that they guarantee soundness and consistency: the specification is sure to mean something. Their disadvantage is that they often result in overspecification: it is not the job of this level of specification to describe or overly constrain possible implementations.

The alternative to constructive specifications is the axiomatic style: we simply state the properties that the new operation is to satisfy. The danger is that axioms can introduce inconsistencies: the specification may mean nothing at all. One way to overcome this difficulty is to provide a second, constructive, specification and to prove that it satisfies the axioms of the first. This differs from the purely constructive approach in that exhibition of a constructively defined model merely serves to demonstrate the consistency of the axiomatization, it is not a prescriptive part of the specification.

In our opinion, a specification language should support both the constructive and axiomatic specification styles. Constructive specifications should satisfy a definitional principle that guarantees they do not introduce inconsistencies (this can require theorem-proving in the case of recursive definitions), and it should also be possible to demonstrate the consistency of axiomatic specifications by exhibiting a constructive model of the axioms.

A specification language is more than a logic; it must also provide mechanisms for structuring and organizing specifications into manageable units, such as modules. Once a module structure is provided, it becomes necessary to consider the visibility of names and the rules for importing and exporting fragments of specification from one context to another. Modules should be parameterized to support re-use and we have found it convenient to use this mechanism to provide some of the benefits that LCF-style systems obtain by polymorphic type-inference. For example, we may have a "lists" module that is parameterized by a type `T`. The type `list`, and functions such as `reverse` that operate on lists, will then be defined with respect to the generic type `T`. Specific instances of the `list` type and `reverse` function are obtained by instantiating the module parameter `T` appropriately. This form of polymorphism at the level of modules may seem less sophisticated than the expression-level polymorphism and type-inference of LCF but, when enriched by the ability to overload names, we have found it adequately expressive and convenient in practice.

It is often necessary to place semantic constraints on the actual parameters that can be supplied to parameterized specification modules. For example, a module specifying the general principle of Noetherian induction should only be applied to a well-founded ordering. The parameters to the module will be a type `T` and a relation $<$ on that type, and the module constraints must express a condition that ensures the relation is a well-founded ordering. One way to specify this (in the language of EHDM [25]) uses a *measure* function:

```
Noetherian: MODULE [T: TYPE, <: [T,T-> bool]]
ASSUMING

measure: VAR [T -> nat]
a, b: VAR T

well_founded: FORMULA (∃ measure : a < b ⊃ measure(a) < measure(b))
```

Any instantiation of this module (for example with the natural numbers supplied for `T` and the successor relation on the naturals supplied for $<$) will cause the corresponding instantiation of the module constraint to be established as a proof obligation similar to a type-correctness condition.

The mechanical support provided for a specification language must include a parser and a typechecker so that specifications can be checked for syntactic and semantic consistency. Since specifications can get quite large, browsers, cross-reference generators, and other similar support tools for navigating a large body of material are highly desirable. In fact, a fully-fledged formal verification system should provide many of the services of a programming support environment, with specification libraries, version management, and so on; there is little point in expending great effort on sound theorem proving if changes to the specification are not tracked.

Since specifications are used to communicate information among the design team, and between the design team, customer, certifying authority, and others, facilities should be provided that allow specifications to be presented in an attractive form that facilitates comprehension. For example, EHDM's cross-reference generator provides typeset text automatically, and its table-driven LaTeX-printer converts specification text such as

```
abs(c(p, i, T) - c(q, i, T))
```

into the typeset notation of conventional mathematics:

$$|c_p^{(i)}(T) - c_q^{(i)}(T)|.$$

The value of such simple tools should not be underestimated.

4 Verification and Theorem Proving

Mechanically-checked verification requires a theorem prover or proof checker. (A proof checker is simply a theorem prover that requires more human guidance.) It might seem that the more powerful and automatic the theorem prover the better. This is a view that needs severe qualification however.

First, to derive the maximum value from the exercise, we want to obtain a genuine proof—that is a chain of argument that will convince a human reviewer—rather a mere grunt of assent from a mechanical theorem prover. It is the proof that distills the insight into why a certain design does its job, and it is the 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 proof, and such methods are unacceptable on this count.

Second, many of the theorems that we wish to prove will turn out to be incorrect—i.e., they will not be theorems after all. It is precisely because the details of unverified proofs can be missing or flawed that mechanical verification is of value: but this means that 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. Powerful automatic theorem provers are generally built on the assumption that they will be put to work on true theorems. When confronted with false theorems, they may relapse into inscrutable silence. Yet it is in the discovery and isolation of mistakes that formal verification can be of most value. The philosopher Lakatos argues similarly for the role of proof in mathematics. Roughly, this view is that successful completion is among the least interesting and useful outcomes of a proof attempt; 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:

> "The virtue of a logical proof is not that it compels belief but that it suggests doubts" [16, page 48].

These limitations on the utility of powerful automatic theorem proving in the large do not apply in the small. Routine manipulations of arithmetic, equalities, disequalities, and inequalities can, and should, be completely mechanized. This is so because a human reviewer does not need to examine the argument for elementary deductions such as $x \geq y \geq z \supset x + 1 > z$, and because *complete* decision procedures are available (at least for the ground case [30]), so that their behavior is predictable. It is hard to overestimate the contribution of the decision procedures in EHDM towards the successful completion of the verifications that we have performed. Conversely, it is enormously tedious to perform verifications involving even modest quantities of arithmetic in systems such as HOL that lack decision procedures.

The key design choices in theorem proving concern the reconciliation of powerful low-level automation with effective human guidance. Truly productive theorem proving requires a symbiosis between man and machine: the user should guide the overall strategy and provide the key insights, and the machine should do the straightforward calculations. The tactics and tacticals of HOL and other LCF-style systems provide the inspiration for the methods of human guidance that we favor. But our primitive inferences are much more powerful than those of HOL and the overall process of proof is orders of magnitude faster. The selection and implementation of relatively few, but powerful, low-level primitives for theorem proving is a major design challenge, since they need to combine together effectively. For example, it is not enough to have decision procedures for arithmetic and separate facilities for conditional rewriting; the conditional rewriter must be able to use the decision procedures in an effective way.

The outcome of a successful proof attempt should be two proof descriptions (or possibly one description that serves two purposes): one should be very close to a proof that can accompany a journal-style presentation of the verification for human consumption; the other should be a description that guides the prover to repeat the proof without human guidance. The latter will be needed in order to rerun the proof at later stages in the overall verification, when many surrounding details may have changed slightly. Ideally, this proof description should be robust—describing a strategy rather than a line by line argument—so that unimportant changes to the specification of lemmas will not derail it. The rerunning of proofs is essential if one regards the purpose of mechanically-checked verification as the acquisition of insight rather than mere certification. As a verification develops, one discovers simplifications, improvements, and generalizations that should be assisted, not discouraged, by investment in an existing verification. Even those most interested in certification should recognize that a specification is seldom static: a change in external requirements or in the specification of an assumed service will require reverification.

The dynamic and creative nature of verification development argues strongly for the ability to perform proofs in any order. Some systems require, or strongly encourage, a bottom up development in which only previously proven lemmas can be cited in a new proof. This is not the way mathematics is performed: one generally prefers to know whether a proposed lemma is adequate to its intended use before attempting to prove it. If proofs can be attempted in any order, it becomes necessary to provide an analysis tool that examines the macroscopic structure of a complete verification in order to ensure the absence of circularities and unproved lemmas. EHDM includes such a tool, which has the additional benefit of providing a complete enumeration of the assumptions, definitions, and axioms on which the verification ultimately depends. This enumeration identifies the requirements that must be satisfied at the next lower level of design, and the real-world assumptions that must be validated empirically.

5 Conclusion

The utility and cost-effectiveness of formal methods can be enhanced by careful selection of the problems to be addressed, and by appropriate choices in the design of specification languages and theorem provers. We have identified some of the important choices in these regards in the previous sections. The two principal issues are the need to reconcile the desire for expressiveness in the specification language with the ability to provide effective mechanical support, and the requirement for the theorem prover to be specialized towards the requirements of verification. The former requires careful selection of the logic and type-system, and delicate design of the additional machinery, such as modules and parameterization. The latter requires the theorem prover to assist in the rapid identification of the sources of errors in incorrect theorems as well as in the certification of true theorems, to produce a proof suitable for human review, to have powerful capabilities for low-level reasoning in standard theories (e.g., arithmetic), and to allow productive interaction with a human user.

No current system for formal specification and verification satisfies all these requirements simultaneously. However, we believe that "next generation" systems developed from HOL, EHDM, and others will come close to doing so. We are actively developing our PVS system toward that end.

Acknowledgments

SRI's verification systems are the collective work of several people and it is a pleasure to acknowledge the creativity and skill contributed by Judy Crow, David Cyrluk, Friedrich von Henke, Sam Owre, and Natarajan Shankar.

References

[1] Jon Barwise. Mathematical proofs of computer system correctness. *Notices of the American Mathematical Society*, 36:844–851, September 1989.

[2] William R. Bevier. Kit and the short stack. *Journal of Automated Reasoning*, 5(4):519–530, December 1989.

[3] William R. Bevier, Warren A. Hunt, Jr., J Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

[4] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, May 1988.

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

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

[7] Frederick P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[8] Ricky W. Butler and Jon A. Sjogren. Formal design verification of digital circuitry. *Reliability Engineering and System Safety*, 32:67–93, 1991.

[9] 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.

[10] Norman Delisle and David Garlan. A formal specification of an oscilloscope. *IEEE Software*, 7(5):29–36, September 1990.

[11] B. Elspas, M. Green, M. Moriconi, and R. Shostak. A JOVIAL verifier. Technical report, Computer Science Laboratory, SRI International, January 1979.

[12] S. Feferman. Polymorphic typed lambda-calculi in a type-free axiomatic framework. In W. Sieg, editor, *Logic And Computation*, pages 101–136. American Mathematical Society, 1990.

[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] William S. Hatcher. *The Logical Foundations of Mathematics*. Pergamon Press, Oxford, UK, 1982.

[15] Ian Hayes, editor. *Specification Case Studies*. Prentice-Hall International Ltd., Hemel Hempstead, UK, 1987.

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

[17] Leslie Lamport. The temporal logic of actions. Technical Report 57, DEC Systems Research Center, Palo Alto, CA, April 1990. A substantially modified version is available dated January 1991.

[18] 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.

[19] 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.

[20] L. Robinson, K. N. Levitt, and B. A. Silverberg. *The HDM Handbook*. Computer Science Laboratory, SRI International, Menlo Park, CA, June 1979. Three Volumes.

[21] Lawrence Robinson and Karl N. Levitt. Proof techniques for hierarchically structured programs. *Communications of the ACM*, 20(4):271–283, April 1976.

[22] W. W. Royce. Managing the development of large software systems. In *Proceedings WESCON*, August 1970.

[23] John Rushby. Formal specification and verification of a fault-masking and transient-recovery model for digital flight-control systems. Technical Report SRI-CSL-91-3, Computer Science Laboratory, SRI International, Menlo Park, CA, January 1991. Also available as NASA Contractor Report 4384, July 1991.

[24] 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.

[25] 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.

[26] Natarajan Shankar. Mechanical verification of a schematic Byzantine fault-tolerant clock synchronization algorithm. Technical Report SRI-CSL-91-4, Computer Science Laboratory, SRI International, Menlo Park, CA, January 1991. Also available as NASA Contractor Report 4386, July 1991.

[27] R. E. Shostak, R. Schwartz, and P. M. Melliar-Smith. STP: A mechanized logic for specification and verification. In D. Loveland, editor, *6th International Conference on Automated Deduction (CADE)*, volume 138 of *Lecture Notes in Computer Science*, New York, NY, 1982. Springer Verlag.

[28] Robert E. Shostak. On the SUP-INF method for proving Presburger formulas. *Journal of the ACM*, 24(4):529–543, October 1977.

[29] Robert E. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(7):583–585, July 1978.

[30] Robert E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2):351–360, April 1979.

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

[32] Jay M. Spitzen, Karl N. Levitt, and Lawrence Robinson. An example of hierarchical design and proof. *Communications of the ACM*, 21(12):1064–1075, December 1978.

[33] J. M. Spivey, editor. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, 1989.

[34] John H. Wensley et al. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, October 1978.

[35] Phillip J. Windley. *The Formal Verification of Generic Interpreters*. PhD thesis, University of California, Davis, CA, June 1990.