Invited paper presented to joint session of CAV and CADE '96. Appears in proceedings of CAV '96, Springer Verlag LNCS 1102, pp. 169–183, July 1996

Automated Deduction and Formal Methods*

John Rushby

Computer Science Laboratory, SRI International,
Menlo Park, CA 94025, USA

Abstract. The automated deduction and model checking communities have developed techniques that are impressively effective when applied to suitable problems. However, these problems seldom coincide exactly with those that arise in formal methods. Using small but realistic examples for illustration, I will argue that effective deductive support for formal methods requires cooperation among different techniques and an integrated approach to language, deduction, and supporting capabilities such as simulation and the construction of invariants and abstractions. Successful application of automated deduction to formal methods will enrich both fields, providing new opportunities for research and use of automated deduction, and making formal methods a truly useful and practical tool.

1 Introduction

Formal methods are a natural application area for automated deduction—yet, with few exceptions, tools for mainstream formal methods provide little more than rudimentary support for deduction, and few theorem provers find application in formal methods. Model checking and related techniques are gaining acceptance in important specialized areas, but have yet to penetrate the larger field. This disconnect between formal methods and the very technologies that could help increase its utility and appeal is unfortunate, and deserves explanation and remedy.

My opinion is that many techniques for automated deduction (and for simplicity I include model checking under this heading) provide excellent solutions to individual problems, but that formal methods require more integrated approaches to provide solutions that are effective across a broad range of problems. In the following sections, I outline some prototypical applications of formal methods and suggest some of the capabilities required of automated deduction if it is to achieve more widespread use in this area. I discuss these topics under three headings: language, theories, and interaction in the sections that follow. Brief conclusions are presented in Section 5.

* This work was supported by the Air Force Office of Scientific Research under contract F49620-95-C0044 and by the National Science Foundation under contract CCR-9509931. The applications described were undertaken for NASA Langley Research Center under contracts NAS1-18969 and NAS1-20334 and for ARPA through NASA Ames Research Center under contract NASA-NAG-2-891.

2 Language

By *formal methods* I mean the use of techniques derived from mathematical logic for the specification and analysis of computational systems. There are two elements here: *specification*, by which I mean a descriptive activity in which logical notation is valued for its contributions to both the intellectual process of design and the communication of designs, and *analysis*, by which I mean systematic and repeatable methods for deducing properties of specifications and of the designs that they represent. Automated deduction has obvious relevance in the mechanization of analysis, but formal methods practitioners attach great importance to specification and are unwilling to compromise on the convenience of expression provided by a full specification language. To achieve acceptance, it therefore seems necessary that automated deduction should be harnessed to rather rich notations.

To suggest some of the capabilities desired, I outline a typical "requirements specification" for a function in the Space Shuttle's control system called "Jet-Select" [6]. This function is responsible for selecting which of the Shuttle's Reaction Control System (RCS) jets (or thrusters) should be fired in order to accomplish a given translational or rotational acceleration. I will concentrate on the "Vernier/Alt" component for rotation, which can operate in one of two modes: in Vernier mode, only the small "vernier" jets are considered for selection; in Alt (alternative) mode, only the larger "primary" jets are considered. The basic Jet-Select calculations are the same whether in Vernier or Alt mode, except that the six vernier jets are treated singly, while the 38 primary jets are treated in groups. (The primary jets are arranged in 14 groups, each consisting of two, three, or four jets located adjacent to each other and firing in the same direction; only 11 of the 14 groups are useful for rotational maneuvers.) In Vernier mode, Jet-Select chooses up to three individual vernier jets to fire, whereas in Alt mode it selects up to three groups of primary jets, and then selects exactly one jet from each of the chosen groups. (The jets within each group are ranked in a priority order and it is the available jet of highest priority that is fired when its group is selected in Alt mode.) Various vernier jets and groups of primary jets are excluded from consideration in certain submodes (e.g., jets whose plumes extend into the area above the cargo bay are excluded in "low +z" mode) and individual jets may be marked "unavailable" due to failure or by crew selection.

The selection of vernier jets or primary groups is performed by an algorithm known as "max dot-product" (this particular exercise in formalization was undertaken in preparation for introduction of a new algorithm called "min angle"). For each vernier jet and primary group, a table records the rotational velocity vector imparted by firing that jet (or a member of that group) for a standard period. (Actually, there are several tables, parameterized by whether there is a payload attached to the Shuttle's robotic arm, and where the arm is positioned.) The algorithm proceeds by first selecting the vernier jet or primary group whose acceleration has the largest scalar (dot) product with the rotational acceleration vector actually desired; the second and third jets (if required) are similarly selected as those with the second and third largest scalar products, *provided* the dot-product of the second exceeds some fraction t_1 of the first, and that of the third exceeds some fraction t_2 of the second.

The major goal here is to use formal methods to specify the desired functionality as clearly as possible. The role of automated deduction in this example is to contribute to validation of the specification by examining putative "challenge" theorems such as "a failed jet will never be selected."

A good specification for this component of Jet-Select should make clear that the max dot-product algorithm is essentially the same in both Vernier and Alt modes, except that in the former it operates over individual vernier jets, while in the latter it operates over groups of primary jets. This argues for a specification notation that provides parameterized theories so that specification of the same algorithm can be instantiated over these different domains. Although not exemplified by Jet-Select, many applications of formal methods also require parameterized representations for standard computer science data structures such as lists, trees, and arrays.

Next, we can observe that the output of Jet-Select is most naturally considered as a set of jets, and the groups of primary jets are also naturally considered as sets. Thus, our specification notation should incorporate a representation for sets. Most practitioners of formal methods prefer their specification notation to be strongly typed, and this particular application seems to call for subtyping: surely the vernier and primary jets are naturally considered as subtypes of the type of all jets. But then the output of the algorithm will be either a set of vernier jets or a set of primary groups (the latter is then converted to a set of primary jets), whereas the output of Jet-Select as a whole must be a set of jets. Hence, our specification notation must somehow extend the subtyping relation between (for example) vernier jets and all jets to a compatible subtyping relation between the *sets* of such jets.

There are (at least) two ways to specify that the (intermediate) result of Jet-Select should be the set of vernier jets or primary groups satisfying the max dot-product criterion. One way would simply axiomatize the desired property, the other would attempt to represent the algorithm suggested by the informal description (i.e., the iterative selection of the three best jets or groups from among those available). The latter approach might require the specification language to incorporate a treatment of imperative programs. It would also require a way to identify the jet or group in a given set that has the maximum dot-product. For generality, we might like to provide a library axiom defining the maximum of a set to be its largest member with respect to some given ordering. This is most directly accomplished by quantification, but we must ensure that the ordering relation has the appropriate algebraic properties and must take proper care of the case where the set is empty, or risk unsoundness. A specification language should help ensure that these obligations are not overlooked.[2]

Most theorem provers support raw logics that lack the notational conveniences mentioned above. In my experience, it quite hopeless to persuade users

[2] For example, the PVS declaration

$$max(s : \mathrm{setof}[T]) : \{t : T | t \in s \land \forall (x : T) : x \in s \supset (t > x \lor x = t)\}$$

of formal methods (let alone those who are not yet users) to adopt such impoverished notations. To observe that it is perfectly *feasible* to provide a specification for Jet-Select in quite primitive logics (e.g., those without quantification) misses the point—this simply is not what users of formal methods want to do.

Left to their own devices, users of formal methods develop or adopt notations such as B, VDM, RAISE, or Z. These make few concessions to the needs of efficient automated deduction and the tools that have been developed for them provide little more than interactive proof checking unsupported by significant automation (e.g., [8, 10]). I have argued elsewhere [14] that choices made in the designs of these languages (e.g., in the case of Z, set theory with partial functions, and no notion of definition) are inimical to automated deduction, and that really efficient deductive support is therefore unlikely to be forthcoming for them.

One of the challenges to those who would provide automated deduction for formal methods is therefore to contribute to the design of specification languages that combine the felicity of expression desired for formal methods with the possibility of powerfully automated support. Rather than being a limitation on specification language design, I believe that closer integration of language and automated deduction can have a liberating effect—because it makes it possible to contemplate design choices that require theorem proving in typechecking. We have exploited this opportunity to some extent in PVS [12] (where subtyping, for example, can generate proof obligations) but many further opportunities remain.

It is not necessary that the logic supported by a theorem prover should *be* a full specification language, but there must be some translation from the latter to the former. Furthermore, the translation must be maintained during interaction with the prover: it is unlikely to be acceptable if proof of a conjecture expressed in the specification language must be conducted in terms of its translation into the primitives of the underlying logic.

3 Theories

Automated deduction must not only support the rich linguistic capabilities desired in formal methods, but must also provide very effective automation for theories that are commonly encountered.

For illustration, I will use a verification of the Interactive Convergence Algorithm for Byzantine fault-tolerant clock synchronization [9] that Friedrich von Henke and I performed some years ago [15]. The goal is to keep the clocks of distributed processors approximately synchronized, given that good clocks have some bounded drift rate, good processors can read the clocks of other good processors with some small error, and faulty processors and clocks are unconstrained (in particular, they can present conflicting information to different good processors). The clock of processor p is represented by an uninterpreted function $c_p(T)$

generates a proof obligation (to show that the type assigned to the value of *max* is inhabited) that can be discharged only if the set s is nonempty and $>$ is a well-ordering.

from "clock time" to "real time" (both interpreted as real numbers).[3] Clocks are adjusted every R clock time units (this duration is called a "frame" and the start time of the i'th frame is denoted $T^{(i)} = T^{(0)} + iR$), during a "synchronization period" of duration S clock time units occurring at the end of the frame. (The interval of the i'th frame is denoted $R^{(i)} = [T^{(i)}, T^{(i+1)}]$, and the i'th synchronization period is denoted $S^{(i)} = [T^{(i+1)} - S, T^{(i+1)}]$.) The adjustment to clock p for period i is $C_p^{(i)}$ clock time units and the adjusted clock for that period is denoted $c_p^{(i)}(T)$, where $c_p^{(i)}(T) = c_p(T + C_p^{(i)})$.

In the i'th synchronizing period, each processor p obtains an estimate $\Delta_{q\,p}^{(i)}$ of the skew between its clock and that of processor q. A parameter ϵ bounds the error in this estimate as follows.

Assumption A2. *If the clock synchronization conditions (defined below) hold for the i'th period, and processors p and q are nonfaulty through period i, then*

$$|\Delta_{q\,p}^{(i)}| \leq S$$

and

$$|c_p^{(i)}(T' + \Delta_{q\,p}^{(i)}) - c_q^{(i)}(T')| < \epsilon$$

for some time T' in $S^{(i)}$.

The algorithm is defined as follows.

Algorithm ICA. *For all processors p:*

$$C_p^{(i+1)} = C_p^{(i)} + \Delta_p^{(i)},$$

where

$$C_p^{(0)} \quad \textit{is arbitrary,}$$

$$\Delta_p^{(i)} = \left(\frac{1}{n}\right) \sum_{r=1}^{n} \bar{\Delta}_{r\,p}^{(i)}, \qquad \textbf{and}$$

$$\bar{\Delta}_{r\,p}^{(i)} = \textbf{if } |\Delta_{r\,p}^{(i)}| < \Delta \textbf{ then } \Delta_{r\,p}^{(i)} \textbf{ else } 0$$

and Δ is a clock time quantity that is a parameter to the algorithm.

The goal is to achieve the following *clock synchronization conditions*, provided that at most m processors (out of n) are faulty through period i, for real time constant δ and clock time constant Σ that are parameters to the algorithm.

Bounded skew: *If p and q are nonfaulty through period i, then*

$$|c_p^{(i)}(T) - c_q^{(i)}(T)| < \delta$$

for all T in $R^{(i)}$.

[3] A specification language with the ability to distinguish clock time and real time as different "dimensions" of the same type provides valuable additional error checking in these constructions.

Bounded adjustment: *If processor p is nonfaulty through period i, then*

$$|C_p^{(i+1)} - C_p^{(i)}| < \Sigma.$$

These conditions can be achieved, provided several assumptions (concerning, for example, the drift rate ρ of good clocks) are satisfied, together with several constraints on the parameters to the algorithm, such as the following.

Constraint C6. $\delta \geq 2(\epsilon + \rho S) + \dfrac{2m\Delta}{n-m} + \dfrac{n\rho R}{n-m} + \dfrac{n\rho\Sigma}{n-m} + \rho\Delta$

The proof depends on several lemmas, of which the following are among the most important.

Lemma 4. *If the clock synchronization conditions hold for i, processors p, q, and r are nonfaulty through period $i+1$, and $T \in S^{(i)}$, then*

$$|c_p^{(i)}(T) + \bar{\Delta}_{r\,p}^{(i)} - [c_q^{(i)}(T) + \bar{\Delta}_{r\,q}^{(i)}]| < 2(\epsilon + \rho S) + \rho\Delta.$$

Lemma 5. *If the bounded skew clock synchronization condition holds for i, processors p and q are nonfaulty through period $i+1$, and $T \in S^{(i)}$, then*

$$|c_p^{(i)}(T) + \bar{\Delta}_{r\,p}^{(i)} - [c_q^{(i)}(T) + \bar{\Delta}_{r\,q}^{(i)}]| < \delta + 2\Delta.$$

The items of interest here are the theories involved: we have arithmetic expressions and relations involving both real and natural numbers, and both interpreted and uninterpreted function symbols. The ubiquity and complexity of the arithmetic used here are such that it would be intolerable to attempt verification of this algorithm without efficient deductive support for arithmetic. A library of lemmas and rewrite rules will not be adequate to the task: decision procedures are needed. The question then is: decision procedures for which theories? The importance of integer arithmetic is such that some tools for formal methods include decision procedures for Presburger arithmetic—that is the quantified theory of integer linear arithmetic. Since we have real numbers as well, a decision procedure for real closed fields might also seem appropriate. The problem with these choices is that we also have uninterpreted function symbols, which takes us outside these decidable theories. Inspection of various formulas appearing in the presentation of the algorithm shows that only Assumption A2 involves a nested quantifier (for T'), everything else is (implicitly) universally quantified at the outermost level. We can conclude that the quantifier reasoning here is likely to be easy, and we may therefore be prepared to deal with it outside the arithmetic decision procedures (either heuristically, or with user guidance). This will allow us to restrict the arithmetic decision procedures to just the ground case—where the combination of linear arithmetic with uninterpreted function symbols is decidable [4].

My experience with formal methods applications is that this tradeoff in favor of deciding ground theories is always worthwhile, since it allows the different decision procedures to be combined. Some theories, such as arithmetic, equality with uninterpreted function symbols, and arrays[4] are so ubiquitous that decision procedures for their ground cases are essential for all productive work. Decision procedures for additional theories may be highly advantageous for particular classes of applications. For example, our experience with processor verification [17] has shown that the (large) library of rewrite rules used for the theory of bitvectors is the main impediment to effective automation, and we conjecture that a decision procedure for bitvectors would have a dramatic benefit. The development of new decision procedures for theories arising in formal methods is a valuable topic for research.

Important requirements for such decision procedures are the following.

- They must work cooperatively to decide the *combination* of their theories.
- They must deal gracefully with terms outside the decided theory. For example, the theory decided by the *SUP-INF* [16] and similar procedures is ground *linear* arithmetic, but several of the formulas used in clock synchronization contain nonlinear terms (and division). Although the full nonlinear case cannot be decided, it is important to deal with special properties (e.g., commutativity, and "a minus times a minus is a plus") without losing those properties that follow simply by treating nonlinear multiplication as uninterpreted. A similarly effective extension to division is also required. (Notice also that some treatment for the partiality of division by zero is needed; this may require coordination between the specification language and its deductive support—in PVS, for example, division by zero is excluded through type rules that generate proof obligations to show the divisor is nonzero.)
- Their behavior must be predictable. One of the strengths of decision procedures over heuristics is that the user should not have to puzzle over whether the failure to prove a conjecture is due to its falsehood, or an inadequate heuristic. This benefit is lost if the decided theory is not clearly characterized. And although performance is hard to guarantee given the super-exponential complexity of most decision procedures, "black holes" (where a small and apparently simple problem takes an inordinate amount of time) are to be avoided. Because they will form the inner loop of larger procedures, even linear speedups in the performance of decision procedures can have a dramatic impact on overall efficiency; more needs to be known about the relative practical performance of various decision procedures for the same problem, which anecdotal evidence indicates can differ by an order of magnitude or more [4]. Conjectures in formal methods applications often give rise to very large formulas, so it is crucial that decision procedures should be implemented in ways that scale reasonably well (using, for example, structure-sharing techniques similar to those in BDDs[5]).

[4] That is (in PVS notation) $f[(x) := y](z) = \textbf{if } z = x \textbf{ then } y \textbf{ else } f(z)$. This is also known as function updating or overriding.

[5] It goes without saying that propositional reasoning must be implemented very efficiently. Ordered binary decision diagrams (OBDDs) are the natural choice, but the Davis-Putnam procedure and the patented algorithm of Stålmarck [18] may be superior in some applications.

- Expressions that cannot be decided should be simplified. Especially in an interactive environment, it is important that the information presented to the user should be as brief and as simple as possible. But it should also be familiar—that is to say, expressions should retain, to the extent possible, the form they were originally given by the user, and should not be arbitrarily normalized. Simplification should merely eliminate redundancy, so that, for example, $(a+1)-1$, **if** *true* **then** a **else** b, and **if** B **then** a **else** a all become a; it should generally refrain from transformations such as that from $x \times (a+b)$ to $x \times a + x \times b$. One of the great advantages of decision procedures over heuristics is that they are sensitive only to the content and not to the form of expressions, so that syntactic representations can be chosen for the convenience of the user rather than the prover.

With standard theories handled by ground decision procedures, the next candidate for automation is quantifier reasoning. Traditional methods for first-order reasoning, such as resolution, do not extend well to the presence of decided ground theories, and therefore find little application in formal methods. (Also, formal methods often use higher-order quantification.) Fortunately, as noted above, there is generally little nesting or alternation of quantifiers in these applications, so that a combination of specialized and heuristic methods work quite well for the majority of cases (difficult cases then require user guidance). Specialized methods include those for conditional rewriting in the presence of decided theories—the close integration of rewriting with linear arithmetic is the source for much of the effectiveness of Boyer and Moore's provers [3], and similar capabilities are required in any system intended to support formal methods. Matching techniques similar to those used in rewriting can also provide heuristic instantiation for general formulas. However, my experience with PVS is that while its conditional rewriter is almost completely effective (i.e., it rarely fails to find a match if one exists), its heuristic instantiation of lemmas and general quantifier reasoning fails (usually by finding an unproductive match) more often than I would like. More effective methods for quantifier reasoning in these contexts (and for restricted instances of the higher-order case) would be a good topic for research.

Inspection of the formulas for clock synchronization shown earlier suggests that, in addition to arithmetic, propositional, and quantifier reasoning, we will also need induction. Proof that the algorithm maintains the clock synchronization conditions is accomplished using simple induction on the frame index i. Several results on finite summations are also used (a key step in the proof is to split the summation in the definition of $\Delta_p^{(i)}$ into m terms constrained by Lemma 5, and $n-m$ constrained by Lemma 4), and these require bounded induction (i.e., induction over a subrange of the natural numbers) on the recursive function that is used to define summation. Given the need for induction, it might seem that powerful automation for inductive proofs, as provided in several systems, would be beneficial. Unfortunately, these methods have generally been developed for rather restricted (e.g., equational or unquantified) logics, and not for the richer context found here. In the absence of suitable automation, the user may be expected to indicate when induction should be used, and to identify the induction variable or expression (PVS, for example, requires this). It is then relatively straightforward to automate selection and instantiation of the appropriate induction scheme; simple tactics can finish the proof of straightforward lemmas (e.g., those needed here for properties of summations), while more explicit user guidance is be needed in more complex cases (e.g., the main induction here). Many formal methods applications require only a couple of inductions and these simple methods are adequate in these cases. Nonetheless, more automated methods (including those for generalization) would be welcome, and the development of suitable techniques is a good research topic.

3.1 Model Checking

Compared to theorem proving methods, model checking and related techniques (such as state exploration and language inclusion) are becoming rather widely used in formal methods. However, I believe that these techniques currently tend to be used standalone in application domains (such as hardware and protocols) to which they are particularly well-suited, rather than being incorporated into traditional formal methods, or integrated with theorem proving.

For my next example, I describe an experiment undertaken by my colleagues Klaus Havelund and Shankar [7], who applied a combination of finite state exploration, theorem proving, and model checking approaches to a simple protocol. Many larger and more significant problems than this have been examined by finite state enumeration and model checking techniques; what is interesting in this exercise is that it points towards an integration of these techniques with theorem proving, and also highlights some of the areas where further research is needed.

Havelund and Shankar began by reducing the protocol to finite state (by manually assigning explicit small integers as the upper bound on the size of certain data structures) and checking certain safety properties with the Murϕ explicit state exploration system [5]. They next verified these properties for the full protocol by theorem proving in PVS using a traditional invariance argument, but found in the process that the desired invariant had to be strengthened by the addition of many additional conjuncts. These were discovered incrementally during the proof attempt; each new proposed conjunct was checked with Murϕ, added to the invariant, and the evolving proof attempted once more. The whole process was iterated until a sufficiently strong invariant was developed; this eventually comprised 57 conjuncts. Seeking a better approach, they developed a finite-state abstraction of the original protocol, verified (by theorem proving) that it was indeed an abstraction, and then verified properties of the abstraction by model checking.

First, notice that the initial "reduction" to finite state in preparation for examination with Murϕ was a manual and ad-hoc process. This seems typical of finite-state analyses: the original problem is transformed by hand into a form that is acceptable to the available tool. The transformation is usually an aggressive simplification that is adequate for refutation but not for verification—meaning that bugs found in the transformed description are likely to correspond to bugs in the original, but the failure to detect bugs in the former cannot be interpreted as verification of the latter. In the case of the protocol studied in these experiments, the maximum number of messages in a file was arbitrarily set to three: bugs that are manifest only with larger file sizes will not be found by this method.

Next, the direct verification of the full protocol was extremely tedious, as the desired safety property had to be strengthened iteratively until it became an invariant. This process took many weeks, which is clearly unacceptable for general practice. Methods for the systematic—and preferably automated—development of invariants therefore constitute a very worthwhile research topic. Of course, one of the advantages of model checking is that it is largely automatic and does not require the development of such invariants. However, when model checking is used for verification rather than refutation, it is necessary to prove that the finite-state description is a true abstraction of the original specification, and this abstraction proof may itself require invariants. Havelund and Shankar in fact reused 45 of the 57 invariants developed for their protocol in their abstraction proof, so the overall saving in effort was not great in this case. This experience highlights another very fruitful area for research: systematic and automated methods for developing finite-state abstractions. Good results are already known for some special cases [2] and I speculate that integration of these methods with model checking will eventually provide an efficient way to verify properties of infinite-state systems.

There were interesting differences between the "reduced" finite-state description checked with Murϕ and the "abstracted" version that was model checked. In the reduced Murϕ description, a file could comprise 1, 2, or 3 messages; in the abstracted description, the size of the untransmitted portion of the file is chosen from the uninterpreted enumeration `NONE`, `ONE`, and `MANY`. The relation between these different approaches—fixing the size vs. introducing abstraction (and additional nondeterminism)—is worthy of investigation.

Although these experiments indicate several areas where additional research is needed, they also demonstrate some promising directions. First, use of Murϕ to check the plausibility of proposed new invariants is representative of a useful general technique: testing conjectures using some lightweight technique before undertaking a full proof. In formal methods applications, many conjectures are false when first proposed and it is best to discover these falsehoods as early and as cheaply as possible, reserving the investment in a full proof until some confidence has been developed that it is likely to be successful. Lightweight methods generally apply to specific, or reduced, cases of the full specification, and automated assistance for creating these reduced cases is a useful addition to any support environment for formal methods. Apart from finite state enumeration, other lightweight techniques include direct evaluation (for executable specifications), and interactive simulation (for specifications that are not directly executable). The latter methods are usually based on specialized and optimized techniques for automated deduction (e.g., rewriting and enumeration over finite quantifiers).

Second, the combination of theorem proving and model checking in the last of the exercises reported above is representative of a promising direction for integrating powerful, but narrow, techniques into a larger system. For example, model checking in PVS is accomplished using an external decision procedure for Park's μ-calculus. This is extended to a decision procedure for μ-calculus on the hereditarily finite fragment of PVS's type system[6] by encoding their values in propositional variables. The branching time temporal logic CTL is then defined in PVS and its model checking problem is cast as a decision problem in μ-calculus. This allows CTL model checking to be smoothly integrated as a proof procedure in PVS. A benefit of this integration is that model checking is available for any conjecture that has the appropriate semantic attributes, independently of its linguistic representation. For example, a tabular specification construct was recently added to PVS; this was then used to formalize a requirements methodology known as SCR, and model checking was then immediately available for SCR specifications [11].

Interesting challenges for the future are to integrate other highly efficient but narrow procedures into a general purpose framework. Examples include model checking methods for hybrid systems and binary moment diagrams.

4 Interaction with the User

I believe that formal methods can deliver most value when applied to problems where traditional methods are inadequate. All the evidence points two principal sources of failure in complex systems: inadequate understanding of potential interactions, and the intrinsically hard parts of a design. Examples of the former often arise in requirements specification, where it is particularly difficult to anticipate all the interactions among the components of a system and between a system and its environment, particularly when operating in the presence of faults. In the case of Jet-Select, for example, our formalization revealed that certain interactions between error reporting and optimization allowed the possibility of firing a failed jet [6]. Examples of the latter often concern algorithms for concurrent, real time, or fault-tolerant behavior (e.g., cache-coherence or clock-synchronization)—where, again, it is difficult to anticipate all possible interactions—or highly optimized calculations whose correctness rests on a long or complex argument (e.g., SRT division and other efficient floating point algorithms).

A consequence of this observation is that automated deduction in support of formal methods will often be applied to very hard problems. It is, in my view, quite unrealistic to expect that such difficult problems can be solved automatically. The issue, then, is how should the user guide and interact with the process of automated deduction? This raises a dual issue: what information and services can automated deduction provide to the user that will assist in the analysis of very difficult problems?

[6] That is, types built recursively from the Booleans, enumerations, explicit finite subranges of the integers, and records, tuples, predicates, and functions of these.

All interaction between the user and tools for automated deduction can be considered an iteration of the following basic steps. What differs from tool to tool is the relative effort devoted to each step, and the rate of iteration.

1. Decide the procedure to be used at the next step. This can range from coarse decisions of overall strategy ("I'll use SMV") to fine issues of tactics ("instantiate the third variable of formula 3 with the following expression").
2. Transform the current representation of the problem into one that is appropriate for the procedure chosen in the previous step. This may be a major undertaking with pencil and paper (e.g., to reduce an infinite-state protocol specification to a finite-state description in the language of SMV), or it may involve mechanized transformations (including recursive application of this whole activity).
3. Set appropriate switches and dials to tune the selected procedure (e.g., choose a variable ordering for BDDs, a weighting strategy for resolution, or an ordering and orientation of lemmas for Nqthm).
4. Invoke the chosen procedure, contemplate the result returned, and iterate the whole process (sometimes, iterate locally over step 3).

My opinion is that the ability to direct this activity in an efficient and productive manner is largely determined by the predictability of the consequences selected by steps 1 and 3, the quality of information returned in step 4, and the efficiency and repeatability of step 2. The user should be able to select a procedure in step 1 on the basis of a description of what it does, not how it works. Deterministic proof procedures (e.g., elementary transformations such as a case split, or quantifier instantiation) and decision procedures are attractive from this point of view, whereas heuristic procedures are not. By the same token, the switches and dials of step 3 should be minimized, since they generally concern how a proof procedure works, rather than the substance of the conjecture under examination. Few users whose interest is formal methods are willing to learn enough about the workings of a proof procedure that they can master many choices here.

The information returned in step 4 should include the result of applying the proof procedure if it was successful (e.g., "proved," or a list of transformed or new subgoals), and an explanation if it was unsuccessful. Decision procedures and model checkers have a special value in the latter case, because they can often return a counterexample that pinpoints the source of difficulty. The ability to return useful information from failure is particularly important in applications of automated deduction to formal methods because it is to be expected that many conjectures will be false—indeed, the efficient discovery and correction of errors is one of the primary reasons for undertaking formal analysis. For this reason, techniques for automated deduction used in formal methods should not be biased towards successful outcomes—for example, they should not be set up to terminate quickly on success at the expense of taking inordinate time to discover failure.

The whole process of formal analysis will be repeated several times as errors are discovered and the design or its specification are adjusted. But the process is not over once we successfully get to "proved" for the first time. Mechanization allows formal methods to be used to explore and refine designs—just as computational fluid dynamics is used to refine aerofoils. Our verification of clock synchronization, for example, has been modified many times: to improve the proof, to eliminate assumptions, to change the specification so that it connects better with the formalization of another part of the overall fault tolerant architecture, to tighten the bound on synchronization achieved, and to change from a Byzantine fault model to a more complex "hybrid" model [13].

The fact that formal analysis will be repeated many times as a specification is first debugged and then refined has consequences for automated deduction. First, it makes it essential, in my view, that step 2 of the interaction loop described above be automated: as the design and its specification evolve, we should recalculate the "reduced" form required for a particular proof procedure, rather than tinker with the existing one. In particular, for reliability as well as efficiency, I believe that reductions and abstractions from infinite-state to finite-state models should be formalized and mechanized, rather than left as an ad-hoc manual process. Second, the "script" of a proof needs to be recorded in manner that is reasonably robust to small changes in the specification. This argues against conducting and recording proofs in low-level and highly specific terms (e.g., "instantiate formula 3 with `x!1`" where `x!1` is the name of a Skolem constant), since the details may change with the specification. It will be more robust to indicate a procedure (e.g., "use unification to find an instantiation"), or to invoke truly automated deduction (e.g., "finish off the proof using resolution"). Finally, it is important to record dependencies among proofs and specifications, so that the user can speedily answer questions such as "what assumptions does this proof depend on?" and "what proofs may be affected if I change this lemma?"

5 Conclusion: The Need for Integration

The field of automated deduction has developed many powerful techniques that could be applied to formal methods. However, the special character of formal methods applications means that some techniques may need to be adapted to the needs of those applications, (e.g., to return more useful information on failure) and that priorities may be different than in other areas (e.g., decision procedures become more important and first order methods such as resolution may become less so). More importantly, most techniques in automated deduction, and also those related to model checking, tend to be rather brittle "point solutions" that are effective against specific classes of problems, whereas formal methods requires an integrated capability that is effective across a wide range of applications. The research challenge in this area is therefore broadly that of integration: different techniques must work together, different theories must be decided in combination, theorem proving and model checking must cooperate, and the needs and capabilities of efficient automated deduction must influence, and be influenced by, the design of expressive specification languages. Success in this endeavor will enrich both fields, providing a new and exciting application for automated deduction, and making formal methods a truly useful and practical tool for the analysis of interesting real systems.

Acknowledgments

My opinions have formed through many stimulating discussions with my colleagues Judy Crow, David Cyrluk, Klaus Havelund, Friedrich von Henke, Patrick Lincoln, Sam Owre, N. Shankar, and M.K. Srivas, and by experiences using PVS (primarily built by Sam Owre and Shankar) and its predecessors.

References

Papers by SRI authors can generally be retrieved from `http://www.csl.sri.com/fm.html`.

[1] Rajeev Alur and Thomas A. Henzinger, editors. *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[2] Saddek Bensalem, Yassine Lakhnech, and Hassen Saïdi. Powerful techniques for the automatic generation of invariants. In Alur and Henzinger [1], pages 323–335.

[3] 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.

[4] David Cyrluk, Patrick Lincoln, and N. Shankar. On Shostak's decision procedure for combinations of theories. In M. A. McRobbie and J. K. Slaney, editors, *Automated Deduction—CADE-13*, number 1104 in Lecture Notes in Artificial Intelligence, pages 463–477, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[5] David L. Dill. The Murϕ verification system. In Alur and Henzinger [1], pages 390–393.

[6] David Hamilton, Rick Covington, and John Kelly. Experiences in applying formal methods to the analysis of software and system requirements. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 30–43, Boca Raton, FL, 1995. IEEE Computer Society.

[7] Klaus Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe FME '96*, number 1051 in Lecture Notes in Computer Science, pages 662–681, Oxford, UK, March 1996. Springer-Verlag.

[8] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A formal Development Support System*. Springer-Verlag, London, UK, 1991.

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

[10] Mogens Nielsen, Klaus Havelund, Kim Ritter Wagner, and Chris George. The RAISE language, method and tools. *Formal Aspects of Computing*, 1(1):85–114, January–March 1989.

[11] Sam Owre, John Rushby, and Natarajan Shankar. Analyzing tabular and state-transition specifications in PVS. Technical Report SRI-CSL-95-12, Computer Science Laboratory, SRI International, Menlo Park, CA, July 1995. Available, with specification files, from `http://www.csl.sri.com/csl-95-12.html`.

[12] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

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

[14] John Rushby. Mechanizing formal methods: Opportunities and challenges. In Jonathan P. Bowen and Michael G. Hinchey, editors, *ZUM '95: The Z Formal Specification Notation; 9th International Conference of Z Users*, volume 967 of *Lecture Notes in Computer Science*, pages 105–113, Limerick, Ireland, September 1995. Springer-Verlag.

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

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

[17] Mandayam K. Srivas and Steven P. Miller. Formal verification of the AAMP5 microprocessor. In Michael G. Hinchey and Jonathan P. Bowen, editors, *Applications of Formal Methods*, Prentice Hall International Series in Computer Science, chapter 7, pages 125–180. Prentice Hall, Hemel Hempstead, UK, 1995.

[18] Gunnar M. N. Stålmarck. System for determining propositional logic theorems by applying values and rules to triplets that are generated from Boolean formula. United States Patent 5,276,897, January 4, 1994.