AN EMPIRICAL EVALUATION OF AUTOMATED THEOREM PROVERS IN SOFTWARE CERTIFICATION

EWEN DENNEY
RIACS/NASA Ames Research Center
M/S 269-2, Moffett Field, California 94035, USA
edenney@email.arc.nasa.gov
BERND FISCHER
RIACS/NASA Ames Research Center
M/S 269-2, Moffett Field, California 94035, USA
fisch@email.arc.nasa.gov
JOHANN SCHUMANN
RIACS/NASA Ames Research Center
M/S 269-2, Moffett Field, California 94035, USA
schumann@email.arc.nasa.gov
Received ()

We describe a system for the automated certification of safety properties of NASA software. The system uses Hoare-style program verification technology to generate proof obligations which are then processed by an automated first-order theorem prover (ATP). We discuss the unique requirements this application places on the ATPs, focusing on automation, proof checking, traceability, and usability, and describe the resulting system architecture, including a certification browser that maintains and displays links between obligations and source code locations. For full automation, the obligations must be aggressively preprocessed and simplified, and we demonstrate how the individual simplification stages, which are implemented by rewriting, influence the ability of the ATPs to solve the proof tasks. Our results are based on 13 comprehensive certification experiments that lead to 366 top-level safety obligations and ultimately to more than 25,000 proof tasks which have been used to determine the suitability of the high-performance provers DCTP, E-Setheo, E, Gandalf, Otter, Setheo, Spass, and Vampire, and our associated infrastructure. The proofs found by Otter have been checked by Ivy.

Keywords: software certification, automated theorem proving, program synthesis, proof checking, traceability, verification condition generator, Hoare logic

1. Introduction

Software certification aims to show that the software in question achieves a certain level of quality, safety, or security. Its result is a *certificate*, i.e., independently checkable evidence of the properties claimed. Certification approaches vary widely, ranging from code reviews to full formal verification, but the highest degree of confidence is achieved with approaches that are based on formal methods and use logic and theorem proving to construct the certificates.

We have developed a certification approach which uses Hoare-style techniques to demonstrate the safety of aerospace software which has been automatically generated from high-level specifications. Our core idea is to extend the code generator so that it simultaneously generates code *and* the detailed annotations, e.g., loop invariants, that enable a fully automated safety proof. A verification condition generator (VCG) processes the annotated code and produces a set of *safety obligations*, which are provable if and only if the code is safe. An automated theorem prover (ATP) then discharges these obligations and the proofs, which can be verified by an independent proof checker, serve as certificates. This approach largely decouples code generation and certification and is thus more scalable than, e.g., verifying the generator or generating code and complete safety proofs in parallel. The ultimate goal of our work is to indirectly increase trust in the code generator by providing explicit evidence of the safety of the generated programs.

In this paper, we evaluate the extent to which the current generation of ATPs is capable of supporting the formal certification of software. In our view, this covers three main aspects. First, full *automation* is crucial since the practicability of our approach depends on it. Second, the ability to generate proof objects and to carry out *proof checking* is essential to create explicit certificates. Third, the extent to which a prover supports various forms of *traceability* has a significant bearing on the ability of an ATP to create meaningful certificates.

Program certification is a demanding application for ATPs because the number of proof obligations is potentially very large and program verification is generally a hard problem domain. However, in our case there are several factors which make a successful ATP application possible. First, we certify separate aspects of safety and not full functional correctness. This separation of concerns allows us to show non-trivial properties like matrix symmetry but results in more tractable obligations than are required for full functional correctness. Second, the extensions of the code generator are specific to the safety properties to be certified and to the algorithms used in the generated programs. This allows us to fine-tune the annotations which, in turn, also results in more tractable obligations. Third, we take advantage of domain-specific knowledge to aggressively simplify the obligations before they are handed over to the prover.

In this paper, we evaluate three hypotheses. The first hypothesis is that the current generation of high-performance ATPs is—in principle—already powerful enough for practical application in program certification. The second hypothesis is that ATPs can still not be considered entirely as black boxes but require careful integration with the application at hand; in particular, the application must carefully preprocess the proof tasks to make them more tractable. The final hypothesis is that proof checkers for first-order logic have not yet reached the same level of maturity as the ATPs themselves, despite the fact that proof checking is, *prima facie*, conceptually simpler than proof finding.

We have tested our hypotheses by running multiple high-performance provers on seven different versions of the 366 safety obligations resulting from certifying five different safety policies for four different programs—in total more than 25,000 proof tasks per prover. In Section 2 we give an overview of the system architecture, describing the safety policies as well as the generation and preprocessing of the proof tasks. In Section 3, we outline the experimental set-up used to evaluate the theorem provers over a range of different preprocessing levels. The detailed results are given in Section 4; they confirm our first two hypotheses: the provers are generally able to solve the emerging obligations but only after substantial preprocessing. However, for almost all programs and all policies, a few hard obligations remain, and a successful certification (i.e., proof of *all* obligations) can be achieved only after even more tuning. Section 5 then discusses the proof checking experiments, and Section 6 looks at some traceability issues and describes a graphical interface. Finally, Section 7 draws some conclusions.

Conceptually, this paper continues work described before[1,2] but the actual implementation of the certification system has been completely revised and substantially extended. We have expanded the range of both algorithms and safety properties which can be certified; in particular, our approach is now fully integrated with the AUTOFILTER system[3] as well as with the AUTOBAYES system[4] and the certification process is now completely automated. We have also implemented a new generic VCG which can be customized for a given safety policy and which directly processes the internal code representation instead of Modula-2 as in the previous version. All these improvements and extensions to the underlying framework result in a substantially larger experimental basis than reported before. Preliminary versions of the current paper describing smaller experimental evaluations have been published at ESFOR 2004[5] and IJCAR 2004.[6]

Related Work Program verification is a popular application domain for theorem provers; we mention only a few systems here. KIV[7,8] is an interactive verification environment which can use different ATPs but relies heavily on term rewriting and user guidance. Sunrise[9] is a fully automatic system that uses custom-designed tactics in HOL to discharge the obligations. The Bali[10] project uses Isabelle to formalize the type system and operational semantics of Java as well as parts of the Java Virtual machine. This approach can be used to perform bytecode verification and to prove type safety. ESC/Java[11] is an automatic verification system but relies on the user to provide additional information on the program, e.g., loop invariants. Houdini[12] is an automatic annotation assistant developed for ESC/Java which guesses invariants, but a significant amount of user interaction remains. The LOOP project[13] has continued this work with ESC/Java2. It is centered on the LOOP compiler,[14] which contains a VCG for Java code which has been annotated with the Java mark-up language JML,[15] and produces proof obligations for the semi-automatic higher-order theorem prover PVS. As with ESC/Java, however, the need for extensive annotations remains a substantial drawback. Also, since the semantics of the language has been formalized using a so-called shallow embedding (where program terms are represented by their semantic denotation in the theorem prover, rather than as an equivalent syntactic datatype), the proof obligations tend to be unwieldy and quickly overwhelm the theorem prover.

Our approach is also conceptually related to proof-carrying code (PCC).[16] However, PCC works on the machine-code level instead of the source-code level (as we do) and concentrates on very simple safety policies (mainly array-bounds safety) which leads to comparatively simple proof obligations. Hence, PCC is complementary to our approach, and a certifying compiler[17] could be used to ensure that the final compilation step does not compromise the demonstrated safety policies. PCC also spawned an entire cottage industry of proof checkers[18]; however, these tend to use various higher-order logics and are thus not applicable for our purposes.

First-order predicate logic suffices for proving the verification tasks generated from our Hoare-style formulation of safety policies and so they can be handled by first-order ATPs. Over the years, a large number of high-performance provers have been developed, using many different calculi and implementation techniques. A systematic evaluation of the performance of the leading ATPs is carried out annually in the CASC theorem prover competition[19,20] using the TPTP[21,22] benchmark library of problems drawn from various domains, including program verification. One aim of these competitions is to evaluate prover performance over a broad range of examples. However, the library tends to be dominated by problems from more mathematical domains, such as algebra, group theory, geometry, planning, and puzzles. A detailed comparison of automated provers for proof obligations arising in the software reuse domain is described in (Ref. 23, 24). Here, the first-order proof obligations generated by matching components from a software library to VDM specifications are processed by earlier versions of the provers Gandalf, Otter, Spass, and Setheo.

Other approaches to program verification based on static analysis[25] (which trades precision for scalability) or model checking[26] (which typically looks at a different class of problems, such as concurrency errors) are complementary to our use of theorem proving. However, their biggest shortcoming for our purposes is that they are unable to provide explicit evidence of correctness (such as proofs).

2. System Architecture

Our certification tool is built as an extension to the AUTOBAYES[4] and AUTOFILTER[3] program synthesis systems. AUTOBAYES works in the statistical data analysis domain and generates parameter learning programs while AUTOFILTER generates state estimation code based on variants of the Kalman filter algorithm. Figure 1 gives an overview of the overall system architecture. The individual components are described in more detail in the subsequent sections. Both underlying synthesis systems take as input a high-level problem specification and generate code that implements the specification. This process is based on the repeated application of schemas. *Schemas* are generic algorithms which are instantiated in a problem-specific way after their applicability conditions have been proven to hold for the given problem specification. The systems first generate C++-style intermediate code which is then compiled down into any of the different supported languages and runtime environments.

For the certification tool, we extended the schemas such that the synthesis systems generate code that is marked up with annotations relevant to the chosen safety policy. These annotations encode local safety information which is then propagated throughout the program. In the next stage, an analysis is carried out by a VCG applying rules from the safety policy to generate verification conditions which are then simplified by a rewrite system. Finally, certification is achieved by sending these simplified verification conditions to an automated theorem prover and checking the resulting proofs.

The architecture distinguishes between *trusted* and *untrusted* components, shown in dark gray and light gray, respectively. Trusted components *must* be correct because any errors in their results can compromise the assurance given by the system; untrusted compo-

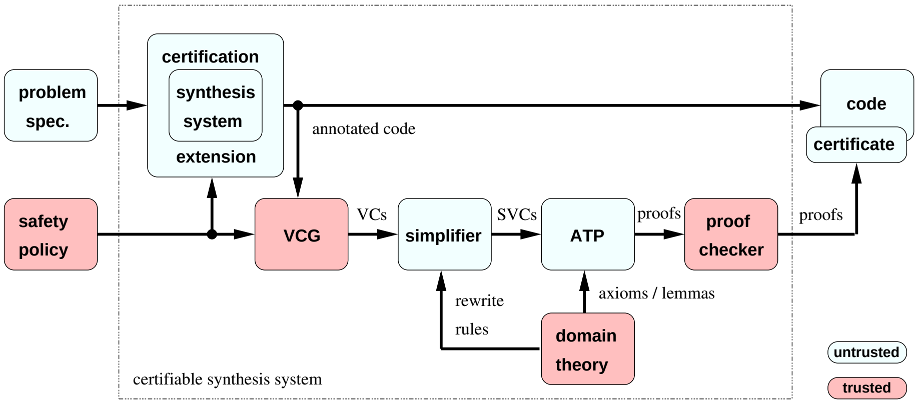

Fig. 1. Certification system architecture.

nents do not affect soundness because their results can be checked by trusted components. In particular, the assurance provided by our certification system does not depend on the correctness of the two largest subsystems: the synthesizer, and the theorem prover; instead, we need only trust the safety policy, the VCG, and the proof checker. This lets us adopt an approach to certification which we call *product-oriented certification*, in contrast to process-oriented approaches, which rely on the qualification (i.e., verification) of the tools being used. A product-oriented approach is more feasible when using complex tools like theorem provers and hence is more scalable.

2.1. *Safety Properties and Safety Policies*

The certification tool automatically certifies that a program satisfies a given *safety property*, i.e., an operational characterization that the program "does not go wrong". It uses a corresponding *safety policy*, i.e., a set of Hoare-style proof rules and auxiliary definitions which are specifically designed to show that programs satisfy the safety property of interest. The distinction between safety properties and policies is explored in related work.[27]

We further distinguish between *language-specific* and *domain-specific* properties and policies. Language-specific properties can be expressed in the constructs of the underlying programming language itself (e.g., array accesses), and are sensible for any given program written in the language. Domain-specific properties typically relate to high-level concepts outside the language (e.g., matrix multiplication), and must thus be expressed in terms of program fragments. Since these properties are specific to a particular application domain, the corresponding policies are not applicable to all programs.

We have defined five different safety properties and implemented the corresponding safety policies. Array-bounds safety (*array*) requires each access to an array element to be within the specified upper and lower bounds of the array. Variable initialization-before-use (*init*) ensures that each variable or individual array element has been explicitly assigned a value before it is used. Both are typical examples of language-specific properties. Matrix symmetry (*symm*) requires certain two-dimensional arrays to be symmetric. Sensor input usage (*inuse*) is a variation of the general *init*-property which guarantees that each sensor reading passed as an input to the Kalman filter algorithm is actually used during the computation of the output estimate. These two examples are specific to the Kalman filter domain. The final example (*norm*) ensures that certain one-dimensional arrays represent normalized vectors, i.e., that their contents add up to one; it is specific to the data analysis domain.

The safety policies can be expressed in terms of two families of definitions. For each command the policy defines a safety condition and a substitution, which captures how the command changes the environmental information relevant to the safety policy. The rules of the safety policy can then be derived systematically from the standard Hoare rules of the underlying programming language.[27]

Table 1. Safety conditions for different safety policies.

Policy	Safety condition	Domain theory
array	$\forall a[i] \in c \cdot a_{lo} \leq i \leq a_{hi}$	arithmetic
init	$\forall$ *read-var* $x \in c \cdot init(x)$	propositional
inuse	$\forall$ *input-var* $x \in c \cdot use(x)$	propositional
symm	$\forall$ *matrix-exp* $m \in c \cdot \forall i, j \,.\, m[i,j] = m[j,i]$	matrices
norm	$\forall$ *vector* $v \in c \cdot \Sigma_{i=v_{lo}}^{v_{hi}} v[i] = 1$	arithmetic, summations

From our perspective here, the safety conditions are the most interesting aspect since they have the greatest bearing on the form of the proof obligations. Table 1 summarizes the different conditions and the domain theories needed to reason about them. Both variable initialization and usage as well as array bounds certification are logically simple and rely just on propositional and simple arithmetic reasoning, respectively, but can require a lot of information to be propagated throughout the program. The symmetry policy needs reasoning about matrix expressions expressed as a first-order quantification over all matrix entries. The vector norm policy is formalized in terms of the summation over entries in a one-dimensional array, and involves symbolic reasoning over finite sums.

2.2. *Generating Proof Obligations*

For certification purposes, the synthesis system *annotates* the code with mark-up information relevant to the selected safety policy. These annotations are part of the schema and are thus instantiated in parallel with the code fragments. The annotations contain local information in the form of logical pre- and post-conditions and loop invariants, which is then propagated through the code. Figure 2 shows as an example code fragment an assignment within a nested loop that is annotated for initialization safety; the annotations shown here include the generated invariants but omit the postconditions.

The fully annotated code is then processed by the VCG, which applies the rules of the safety policy in order to generate the safety conditions. As usual, the VCG works backwards through the code. At each statement, the safety conditions are generated and the

```
for(i = 0; i <= 5; i++)
  /*{ inv forall x,y:int . 0<=x<=i-1 && 0<=y<=5 =>
          tmp2_init[x][y]==init
  }*/
  for(j = 0; j <= 5; j++)
    /*{ inv forall x,y:int . 0<=x<=5 && 0<=y<=5 =>
              (x<i => tmp2_init[x][y]==init) &&
              (x==i && y<j => tmp2_init[x][y]==init)
    }*/
    tmp2[i][j] = id[i][j] - tmp1[i][j];
```

Fig. 2. Generated Code with Annotations.

$$
\begin{array}{ll}
\left.\begin{array}{l}
\ldots\ \forall\, x, y \cdot 0 \le x \le 5 \wedge 0 \le y \le 5 \Rightarrow \mathit{sel}(\mathit{id\_init}, x, y) = \mathit{init} \\
\wedge\ \ \forall\, x, y \cdot 0 \le x \le 5 \wedge 0 \le y \le 5 \Rightarrow \mathit{sel}(\mathit{tmp1\_init}, x, y) = \mathit{init}
\end{array}\right\} & \text{environmental information} \\
\left.\begin{array}{l}
\ldots\ \forall\, x, y \cdot 0 \le x \le i-1 \wedge 0 \le y \le 5 \Rightarrow \mathit{sel}(\mathit{tmp2\_init}, x, y) = \mathit{init} \\
\wedge\ \ \forall\, x, y \cdot 0 \le y \le 5 \wedge 0 \le x \le 5 \Rightarrow \\
\qquad (x < i \Rightarrow \mathit{sel}(\mathit{tmp2\_init}, x, y) = \mathit{init} \wedge \\
\qquad (y < j \wedge x = i \Rightarrow \mathit{sel}(\mathit{tmp2\_init}, x, y) = \mathit{init})))
\end{array}\right\} & \text{invariants} \\
\left.\ldots\ 0 \le i \le 5 \wedge 0 \le j \le 5\right\} & \text{loop bounds} \\
\left.\Rightarrow\ \ (\mathit{sel}(\mathit{id\_init}, i, j) = \mathit{init} \wedge \mathit{sel}(\mathit{tmp1\_init}, i, j) = \mathit{init})\right\} & \text{safety condition}
\end{array}
$$

Fig. 3. Structure of a safety obligation.

safety substitutions are applied. The VCG has been designed to be "correct-by-inspection", i.e., to be sufficiently simple that it is straightforward to see that it correctly implements the rules of the logic. Hence, the VCG does not implement any optimizations, such as structure sharing on verification conditions (VCs) or even apply any simplifications; in particular, it does not actually apply the substitutions but maintains explicit formal substitution terms. Consequently, the generated VCs tend to be large and must be simplified separately; the more manageable simplified verification conditions (SVCs) which are produced are then processed by a first-order theorem prover. The resulting proofs can be sent to a proof checker, e.g., Ivy.[28]

The structure of a typical safety obligation (after substitution reduction and simplification) is given in Figure 3. It corresponds to the code fragment shown in Figure 2. Most of the hypotheses consist of annotations which have been propagated through the code and are, in the best case, merely irrelevant to the line at hand but, in the worst case, prevent the prover from finding a proof. The proof obligation also contains the local loop invariants together with bounds on `for`-loops. Finally, the conclusion is generated from the safety conditions for the statement given by the corresponding safety policy. Although safety obligations with more complex conclusions can arise with the *symm* and *norm* policies, they always have this general form.

2.3. *Processing Proof Obligations and Connecting the Prover*

The simplified safety obligations are exported as a number of individual proof obligations using TPTP first-order logic syntax.[22] A small script then adds the axioms of the domain theory, before the completed proof task is processed by the theorem prover. Parts of the domain theory are generated dynamically in order to facilitate reasoning with (small) integers. The domain theory is described in more detail in Section 3.3.

The connection to a theorem prover is straightforward. For provers that do not accept the TPTP syntax, the appropriate TPTP2X-converter is used before invoking the theorem prover. In the experiments, run-time measurement and prover control (e.g., aborting provers) were performed with the same TPTP tools as in the CASC competition.

3. Experimental Setup

3.1. *Program Corpus*

As a basis for the certification experiments we generated annotated programs from four different specifications which were written prior to and independently of the experiments. The size of the generated programs ranges from 431 to 1157 lines of commented C-code, including the annotations. Table 3 in Section 4 gives a more detailed breakdown. The first two examples are AUTOFILTER specifications. `ds1` is taken from the attitude control system of NASA's Deep Space One mission.[3] `iss` specifies a component in a simulation environment for the Space Shuttle docking procedure at the International Space Station. In both cases, the generated code is based on Kalman filter algorithms, which make extensive use of matrix operations. The other two examples are AUTOBAYES specifications which are part of a more comprehensive analysis[29,30] of planetary nebula images taken by the Hubble Space Telescope. Although these data analysis applications are not safety-critical, they can run on board a spacecraft, thus making the software subject to qualification. `segm` describes an image segmentation problem for which an iterative (numerical) statistical clustering algorithm is synthesized. Finally, `gauss` fits an image against a two-dimensional Gaussian curve. This requires a multivariate optimization which is implemented by the Nelder-Mead simplex method. The code generated for these two examples has a substantially different structure from the state estimation examples. First, the numerical optimization code contains many deeply nested loops. Also, some of the loops are convergence loops which have no fixed upper bounds but are executed until a dynamically calculated error value gets small enough. In contrast, in the Kalman filter code, all loops are executed a fixed (i.e., known at synthesis time) number of times. Second, the numerical optimization code accesses all arrays element by element and contains no operations on entire matrices (e.g., matrix multiplication). The example specifications and all generated proof obligations can be found at `http://ase.arc.nasa.gov/autobayes/ijait`.

3.2. *Simplification*

Proof task simplification is an important and integral part of our overall architecture. However, as observed before,[23,24,31] simplifications—even on the purely propositional level—

can have a significant impact on the performance of a theorem prover. In order to evaluate this impact, we used six different rewrite-based simplifiers to generate multiple versions of the safety obligations. We focus on rewrite-based simplifications rather than decision procedures because rewriting is easier to certify: each individual rewrite step $T \rightsquigarrow S$ can be traced and checked independently, e.g., by using an ATP to prove that $S \Rightarrow T$ holds.

Baseline The baseline for all simplifications is given by the rewrite system $\mathcal{T}_\emptyset$ which eliminates the extra-logical constructs (including explicit formal substitutions) which the VCG employs during the construction of the safety obligations. Our original intention was to axiomatize these constructs in first-order logic and then (ab-) use the provers for this elimination step, but that turned out to be infeasible. The main problem is that the combination of large terms and equality reasoning produces tremendous search spaces.

Propositional Structure The first two proper simplification levels only work on the propositional structure of the obligations. $\mathcal{T}_{\forall,\Rightarrow}$ splits the few but large obligations generated by the VCG into a large number of smaller obligations. It consists of two rewrite rules $\forall x \cdot P \wedge Q \rightsquigarrow (\forall x \cdot P) \wedge (\forall x \cdot Q)$ and $P \Rightarrow (Q \wedge R) \rightsquigarrow (P \Rightarrow Q) \wedge (P \Rightarrow R)$ which distribute universal quantification and implication, respectively, over conjunction. Each of the resulting conjuncts is then treated as an independent proof task. $\mathcal{T}_{\text{prop}}$ simplifies the propositional structure of the obligations more aggressively. It uses the rewrite rules

$$\begin{array}{ll}
\neg\, \mathit{true} \rightsquigarrow \mathit{false} & \neg\, \mathit{false} \rightsquigarrow \mathit{true} \\
\mathit{true} \wedge P \rightsquigarrow P & \mathit{false} \wedge P \rightsquigarrow \mathit{false} \\
\mathit{true} \vee P \rightsquigarrow \mathit{true} & \mathit{false} \vee P \rightsquigarrow P \\
P \Rightarrow \mathit{true} \rightsquigarrow \mathit{true} & P \Rightarrow \mathit{false} \rightsquigarrow \neg\, P \\
\mathit{true} \Rightarrow P \rightsquigarrow P & \mathit{false} \Rightarrow P \rightsquigarrow \mathit{true} \\
P \Rightarrow P \rightsquigarrow \mathit{true} & (P \wedge Q) \Rightarrow P \rightsquigarrow \mathit{true} \\
P \Rightarrow (Q \Rightarrow R) \rightsquigarrow (P \wedge Q) \Rightarrow R & \forall x \cdot \mathit{true} \rightsquigarrow \mathit{true}
\end{array}$$

in addition to the two rules in $\mathcal{T}_{\forall,\Rightarrow}$. The rules have been chosen so that they preserve the overall structure of the obligations as far as possible; in particular, conjunction and disjunction are not distributed over each other and implications are not eliminated. Their impact on the clausifier should thus be minimal.

Ground Arithmetic This simplification level additionally handles common extensions of plain first-order logic, i.e., equality, orders, and arithmetic. The rewrite system $\mathcal{T}_{\text{eval}}$ contains rules for the reflexivity of equality and total orders as well as the irreflexivity of strict (total) orders, although the latter rules are not invoked on the example obligations. In addition, it normalizes orders into $\leq$ and $>$ using the rules

$$\begin{array}{ll}
x \geq y \rightsquigarrow y \leq x & \neg\, x > y \rightsquigarrow x \leq y \\
x < y \rightsquigarrow y > x & \neg\, x \leq y \rightsquigarrow x > y
\end{array}$$

Since the programs and thus the generated safety obligations contain occurrences of the different symbols, these eliminations have to be applied explicitly by the simplifier. However, the choice of the specific symbols is to some extent arbitrary; choosing for example $<$ instead of $>$ makes no difference. We could even replace the two rules on the right with a single rule $x > y \leadsto \neg\, x \leq y$ and thus eliminate all but one ordering symbol but instead decided to minimize the term size rather than the signature size.

$\mathcal{T}_{\text{eval}}$ also contains rules to evaluate ground integer operations (i.e., addition, subtraction, and multiplication), equalities, and partial and strict orders. Moreover, it converts addition and subtraction with one small integer argument (i.e., $n \leq 5$) into Pressburger notation, using rules of the form $n + 1 \leadsto succ(n)$ and $n - 1 \leadsto pred(n)$. For many safety policies (e.g., *init*), terms of this form are introduced by relativized bounded quantifiers (e.g., $\forall x \cdot 0 \leq x \leq n - 1 \Rightarrow P(x)$) and contain the only occurrences of arithmetic operators. A final group of rules handles the interaction between *succ* and *pred*, as well as with the orders.

$$\begin{array}{ll} \mathit{succ}(\mathit{pred}(x)) \leadsto x & \mathit{pred}(\mathit{succ}(x)) \leadsto x \\ \mathit{succ}(x) \leq y \leadsto x < y & \mathit{succ}(x) > y \leadsto x \geq y \\ x \leq \mathit{pred}(y) \leadsto x < y & x > \mathit{pred}(y) \leadsto x \geq y \end{array}$$

Language-Specific Simplification The next level handles constructs which are specific to the program verification domain, in particular array-expressions and conditional expressions, encoding the necessary parts of the language semantics. The rewrite system $\mathcal{T}_{\text{array}}$ adds rewrite formulations of McCarthy's array axioms,[32] i.e., $\mathit{sel}(\mathit{upd}(a, i, v), j) \leadsto i = j\ ?\ v : \mathit{sel}(a, j)$ for one-dimensional arrays and similar forms for higher-dimensional arrays. Some safety policies are formulated using arrays of a given dimensionality which are uniformly initialized with a specific value. These are represented by a *constarray*-term, for which similar rules are required, e.g., $\mathit{sel}(\mathit{constarray}(v, d), i) \leadsto v$.

Nested *sel*/*upd*-terms, which result from sequences of individual assignments to the same array, lead to nested conditionals which in turn lead to an exponential blow-up during the subsequent normalization step. $\mathcal{T}_{\text{array}}$ thus also contains two rules $\mathit{true}\ ?\ x : y \leadsto x$ and $\mathit{false}\ ?\ x : y \leadsto y$ to evaluate conditionals.

In order to properly assess the effect of these domain-specific simplifications, we also experimented with a rewrite system $\mathcal{T}_{\text{array*}}$, which applies the two *sel*-rules in isolation.

Policy-Specific Simplification The most aggressive simplification level $\mathcal{T}_{\text{policy}}$ uses a number of rules which are fine-tuned to handle specific situations that frequently arise with the individual safety policies. The *init*-policy uses a rule

$$\forall x \cdot 0 \leq x \leq n \Rightarrow (x \neq 0 \wedge \ldots \wedge x \neq n \Rightarrow P) \leadsto \mathit{true}$$

which is derived from the finite induction axiom to handle the result of simplifying nested *sel*/*upd*-terms. For *inuse*, we need a single rule $\mathit{def} = \mathit{use} \leadsto \mathit{false}$, which follows from the fact that the two tokens *def* and *use* used by the policy are distinct. For *symm*, we make use of a lemma about the symmetry of specific matrix expressions: $A + BCB^T$ is already symmetric if (but not only if) the two matrices A and C are symmetric, regardless of the symmetry of B. The rewrite rule

$$\begin{array}{l} \mathit{sel}(A + BCB^T, i, j) = \mathit{sel}(A + BCB^T, j, i) \\ \leadsto \mathit{sel}(A, i, j) = \mathit{sel}(A, j, i) \wedge \mathit{sel}(C, i, j) = \mathit{sel}(C, j, i) \end{array}$$

formulates this lemma in an element-wise fashion.

For the *norm*-policy, the rules become a lot more specialized and complicated. Two rules are added to handle the inductive nature of finite sums:

$$\begin{aligned}
&\textstyle\sum_{i=0}^{\mathit{pred}(0)} x \leadsto 0\\
&P \wedge x = \textstyle\sum_{i=0}^{\mathit{pred}(n)} Q(i) \Rightarrow x + Q(n) = \textstyle\sum_{i'=0}^{n} Q(i')\\
&\;\leadsto P \wedge x = \textstyle\sum_{i=0}^{\mathit{pred}(n)} Q(i) \Rightarrow \textstyle\sum_{i=0}^{n} Q(i) = \textstyle\sum_{i=0}^{n} Q(i)
\end{aligned}$$

The first rule directly implements the base case of the induction; the second rule, which implements the step case, is more complicated. It requires alpha-conversion for the summations as well as higher-order matching for the body expressions, both of which are, however, under explicit control of this specific rewrite rule and not the general rewrite engine, and are implemented directly as Prolog predicates. Note that the right hand side can easily be simplified into *true* by the application of further rules. A similar rule is required in a very specific situation to substitute an equality into a summation:

$$\begin{aligned}
&P \wedge (\forall i \cdot 0 \le i \le n \Rightarrow x = \mathit{sel}(f,i)) \Rightarrow \textstyle\sum_{i=0}^{n} \mathit{sel}(f,i) = 1\\
&\leadsto P \wedge (\forall i \cdot 0 \le i \le n \Rightarrow x = \mathit{sel}(f,i)) \Rightarrow \textstyle\sum_{i=0}^{n} x = 1
\end{aligned}$$

The above rules capture the central steps of some of the proofs for the *norm*-policy and mirror the fact that these are essentially higher-order inferences.

Another set of rewrite rules handles all occurrences of the random number generator by asserting that the number is within its given range, i.e., $l \le \mathit{rand}(l,u) \le u$. Since no other property of random numbers is used, *rand* is treated as an uninterpreted function symbol.

Normalization The final preprocessing step transforms the obligations into pure first-order logic. It eliminates conditional expressions which occur as top-level arguments of predicate symbols, using rules of the form $P \mathrel{?} T : F = R \leadsto (P \Rightarrow T = R) \wedge (\neg P \Rightarrow F = R)$ and similarly for partial and strict orders. A number of congruence rules move nested occurrences of conditional expressions into the required positions. Finite sums, which only occur in obligations for the *norm*-policy, are represented with a de Bruijn-style variable-free notation.

Table 2. Number of rewrite rules used in consecutive phases of different simplifications.

	$\mathcal{T}_\emptyset$	$\mathcal{T}_{\forall,\Rightarrow}$	$\mathcal{T}_{\text{prop}}$	$\mathcal{T}_{\text{eval}}$	$\mathcal{T}_{\text{array}}$	$\mathcal{T}_{\text{array*}}$	$\mathcal{T}_{\text{policy}}$
simplification	N/A	3	17	42	42	2	61
normalization	8	8	8	8	8	8	8
clean-up	N/A	N/A	N/A	N/A	31	3	31

Control The simplifications are performed by a small but reasonably efficient rewrite engine implemented in Prolog (cf. Table 3 for runtime information). This engine does not support full AC-rewriting but flattens and orders the arguments of AC-operators. The rewrite rules, which are implemented as Prolog clauses, then do their own list matching but can take the list ordering into account. The rules within each system are applied exhaustively. However, the two most aggressive simplification levels $\mathcal{T}_{\text{array}}$ and $\mathcal{T}_{\text{policy}}$ are followed by a structural "clean-up" phase. This consists of the normalization followed by the propositional simplifications $\mathcal{T}_{\text{prop}}$ and the finite induction rule. Similarly, $\mathcal{T}_{\text{array*}}$ is followed by the normalization and then by $\mathcal{T}_{\forall,\Rightarrow}$ to split the obligations. Table 2 shows the number of rewrite rules for each simplification level, as well as for normalization and clean-up.

3.3. *Domain Theory*

Each safety obligation is supplied with a first-order domain theory. In our case, the domain theory consists of a fixed part which contains 44 axioms, and a set of axioms which is generated dynamically for each proof task. The static axioms define the usual properties of equality and the order relations, as well as axioms for Pressburger arithmetic and for the domain-specific operators (e.g., array accesses and matrix operations). This part axiomatizes 22 different predicate and function symbols. The dynamic axioms are added to avoid the generation of large terms of the form $succ(\ldots succ(0)\ldots)$. They are required because most theorem provers cannot calculate with integers. For all different integer literals n, m occurring in the proof task, we generate the corresponding axioms of the form $m > n$. For small integers (i.e., $n \leq 5$), we also generate axioms for explicit successor-terms, i.e., $n = succ(\ldots succ(0)\ldots)$ and add a finite domain schema of the form $\forall x \cdot 0 \leq x \leq n \Rightarrow (x = 0 \vee x = 1 \vee \ldots \vee x = n)$. In our application domain, these axioms are needed for some of the matrix operations; thus n can be limited to the statically known maximal size of the matrices. The default set of axioms contains all the formulas required for each of the safety policies. A second, reduced domain theory omits axioms that are only relevant to a specific policy. This is described and evaluated in Section 4.5.

3.4. *Theorem Provers*

For the experiments, we selected several high-performance theorem provers for untyped first-order formulas with equality. Most of the provers participated in the the CASC-19[19] or CASC-J2[20] prover competitions and ranked highly in the FOL or MIX categories. In the experiments, we used the default parameter settings or the competition parameter settings and none of the special features of the provers. None of the provers were tuned in any way for this set of proof tasks, with the exception of Otter, where the developer provided an alternative parameter setting since the defaults proved unsuitable. The ATPs were given first-order formulas in TPTP syntax as input when applicable; we then relied on their respective built-in clausification modules, unless otherwise noted. However, ATPs that accept only clausal normal form (i.e., DCTP, Gandalf, and Otter) were provided with the results of the TPTP clausifier as their input. For each proof obligation, we limited the run-time to 60 seconds; the CPU time actually used was measured with the TPTP tools on a 2.4GHz standard Linux PC with 4GB memory.

For DCTP[33] we used version 10.21p with the default parameter settings, which proved overall to be marginally better than the CASC settings. We experimented with six different variants of E-Setheo.[34] Two variants were derived from the CASC-19 version. For E-Setheo-03F, we used Flotter V2.1[35,36] instead of the built-in TPTP clausifier to convert the formulas into a set of clauses. E-Setheo-03n is a development snapshot with several minor improvements over the original CASC-19 version. We also used the CASC-J2 version E-Setheo-04 "as-is", from which we derived two further variants E-Setheo-04F and E-Setheo-04T by replacing E's clausifier with Flotter V2.1 and the TPTP clausifier, respectively. Finally, we used an "E-free" variant of E-Setheo by removing E from all schedules; this is denoted here by Setheo but note that this does not correspond to any of the earlier Setheo-versions (i.e., before version 3.3). For E,[37] we used version 0.82 which was obtained directly from the developer. For Gandalf,[38] we used version c-2.6. The clausal normal form produced by the TPTP clausifier was post-processed so that the set of support only contained clauses originating in the actual proof task, while all clauses originating from the axioms were put into the `list(usable)` section. We also used a variant Gandalf-F, in which the TPTP clausifier was replaced by Flotter V2.1. Spass 2.1 was obtained from the developer's website.[35] For Vampire,[39] we used the three most recent CASC competition versions (Vampire 5.0, Vampire 6.0, and Vampire 7.0), taken directly "out of the box." For comparison purposes, we also used Otter V3.0.6,[40] which is from April 2000 but has been essentially unchanged since 1996.

4. Empirical Results

4.1. *Generating and Simplifying Obligations*

Table 3 summarizes the results of generating the different versions of the safety obligations. For each of the example specifications, it lists the size $|\,P\,|$ of the generated programs (without annotations), the applicable safety policies, the size $|\,A\,|$ of the generated annotations, and then, for each simplifier, the number N of generated obligations and the elapsed time T. The elapsed times include synthesis of the programs as well as generation, simplification, and file output of the safety obligations; synthesis alone accounts for approximately 90% of the times listed under the *array* safety policy. In general, the times for generating and simplifying the obligations are moderate compared to both generating the programs and discharging the obligations. All times are CPU times and have been measured in seconds using the Unix `time` command.

Almost all of the generated obligations are valid, i.e., the generated programs are safe. The only exception is the *inuse*-policy which produces one invalid obligation for each of the `ds1` and `iss` examples. This is a consequence of the original specifications which do not use all elements of the initial state vectors. The invalidity is confined to a single conjunct in one of the original obligations, and since none of the rewrite systems contains a distributive law, the number of invalid obligations does not increase with simplification.

The first four simplification levels show the expected results. The baseline $\mathcal{T}_\emptyset$ yields relatively few but large obligations which are then split up by $\mathcal{T}_{\forall,\Rightarrow}$ into a much larger (on average more than an order of magnitude) number of smaller obligations. The next two levels then eliminate a large fraction of these obligations. Here, the propositional simplifier $\mathcal{T}_{\text{prop}}$ alone already discharges between 50% and 90% of the obligations while the additional effect of evaluating ground arithmetic ($\mathcal{T}_{\text{eval}}$) is much smaller and generally well below 25%. The only significant difference occurs for the *array*-policy where more than 80% (and in the case of `ds1` all) of the remaining obligations are reduced to true. This is a consequence of

Table 3. Generation of safety obligations.

Example	$\|P\|$	Policy	$\|A\|$	$\mathcal{T}_{\emptyset}$ N	$\mathcal{T}_{\emptyset}$ T	$\mathcal{T}_{\forall,\Rightarrow}$ N	$\mathcal{T}_{\forall,\Rightarrow}$ T	$\mathcal{T}_{\text{prop}}$ N	$\mathcal{T}_{\text{prop}}$ T	$\mathcal{T}_{\text{eval}}$ N	$\mathcal{T}_{\text{eval}}$ T	$\mathcal{T}_{\text{array}}$ N	$\mathcal{T}_{\text{array}}$ T	$\mathcal{T}_{\text{array}^*}$ N	$\mathcal{T}_{\text{array}^*}$ T	$\mathcal{T}_{\text{policy}}$ N	$\mathcal{T}_{\text{policy}}$ T
`ds1`	431	*array*	0	11	5.5	103	5.3	55	5.4	1	5.5	1	5.5	103	5.6	1	5.5
		init	87	21	9.5	339	14.1	150	11.3	142	11.0	74	10.5	543	20.1	74	11.4
		inuse	61	19	7.3	453	12.9	59	7.7	57	7.6	21	7.4	682	16.2	21	8.1
		symm	75	17	4.8	101	5.7	21	4.7	21	4.9	858	66.7	2969	245.6	865	70.8
`iss`	755	*array*	0	1	24.6	582	28.1	114	24.8	4	24.2	4	24.0	582	27.9	4	24.7
		init	88	2	39.5	957	65.9	202	42.3	194	41.8	71	39.2	1378	82.6	71	39.7
		inuse	60	2	33.4	672	68.1	120	36.7	117	35.7	28	32.6	2409	79.1	1	31.6
		symm	87	1	33.0	185	34.9	35	28.1	35	27.9	479	71.0	3434	396.8	480	66.2
`segm`	517	*array*	0	29	3.0	85	3.3	8	2.9	3	2.9	3	3.0	85	3.3	1	3.0
		init	171	56	6.5	464	12.1	172	7.8	130	7.7	121	7.6	470	12.8	121	7.6
		norm	195	54	3.8	155	5.0	41	3.8	30	3.6	32	3.8	157	5.2	14	3.6
`gauss`	1039	*array*	20	69	21.0	687	24.9	98	21.2	20	21.0	20	20.9	687	24.3	20	21.3
		init	118	85	49.8	1417	65.5	395	54.1	324	53.2	316	53.9	1434	66.2	316	54.3

the large number of obligations which have the form $\neg(n \leq n) \Rightarrow P$ for an integer constant n representing the (lower or upper) bound of an array. The effect of the domain-specific simplifications is at first glance less clear. Using the array-rules only ($\mathcal{T}_{\text{array}^*}$) generally leads to an increase over $\mathcal{T}_{\forall,\Rightarrow}$ in the number of obligations; this is even greater than an order of magnitude for the *symm*-policy. However, in combination with the other simplifications ($\mathcal{T}_{\text{array}}$), most of these obligations can be discharged again, and we generally end up with fewer obligations than before; again, the *symm*-policy is the only exception. The effect of the final policy-specific simplifications is, as should be expected, highly dependent on the policy. For *inuse* and *norm* a further reduction is achieved, while the rules for *init* and *symm* only reduce the size of the obligations.

4.2. *Running the Theorem Provers*

Table 4 summarizes the results obtained from running the theorem provers on all proof obligations (except for the two invalid obligations from the *inuse*-policy). For the "prover families" E-Setheo and Vampire, however, the table only contains entries for the best variants, E-Setheo-03n and Vampire 6.0, respectively. The results are grouped by the different simplification levels. Each line in the table corresponds to the proof tasks originating from a specific safety policy (*array*, *init*, *inuse*, *symm*, and *norm*). Then, for each prover, the percentage of solved proof obligations and the total CPU time are given. Note that T_{ATP} also includes the actual CPU times for failed proof attempts.

For the fully simplified version ($\mathcal{T}_{\text{policy}}$), all provers are able to find proofs for all tasks originating from at least one safety policy; E-Setheo-03F can even discharge *all* the emerging safety obligations. This result is central for our application since it shows that current ATPs can in fact be applied to certify the safety of synthesized code, confirming our first hypothesis.

For the unsimplified safety obligations, however, the picture is quite different. Here, the provers can only solve a relatively small fraction of the tasks and leave an unaccept-

Table 4. Theorem prover results: Percentage of solved proof tasks and total ATP-times.

			DCTP		E-Setheo		E		Gandalf		Otter		Spass		Vampire	
$\mathcal{T}$	Policy	N	%	T_{ATP}	%	T_{ATP}	%	T_{ATP}	%	T_{ATP}	%	T_{ATP}	%	T_{ATP}	%	T_{ATP}
$\mathcal{T}_{\emptyset}$	*array*	110	95.5	305	97.3	240	98.2	125	97.3	215	85.5	972	99.1	75	98.2	170
	init	164	8.5	9000	7.9	9068	20.7	7860	23.2	8167	6.7	9183	72.6	2753	8.5	9004
	inuse	19	26.3	840	47.4	660	52.6	612	68.4	541	42.1	678	68.4	486	57.9	603
	symm	18	16.7	900	16.7	902	16.7	900	16.7	900	16.7	900	50.0	612	16.7	900
	norm	54	61.1	1260	61.1	1281	63.0	1213	68.5	1082	31.5	2222	72.2	927	61.1	1264
$\mathcal{T}_{\forall,\Rightarrow}$	*array*	1457	99.9	161	99.9	960	99.9	132	99.9	496	99.5	776	99.9	148	99.9	206
	init	3177	93.1	13514	97.9	10465	97.8	5153	98.1	4722	90.9	18545	96.8	8927	94.4	15639
	inuse	1123	91.9	5576	97.2	2823	97.9	1782	97.5	2063	96.6	2503	98.8	1630	94.3	4531
	symm	286	89.9	1775	93.3	4186	90.2	1783	90.5	1710	80.8	3501	89.9	1859	87.8	2968
	norm	155	90.3	904	90.3	983	90.3	906	90.3	904	76.1	2246	90.3	915	89.0	1030
$\mathcal{T}_{\text{prop}}$	*array*	275	99.3	147	99.3	262	99.3	126	99.3	301	97.1	639	99.3	154	99.3	183
	init	919	78.0	12454	92.7	5499	92.4	4448	93.2	4718	76.9	13764	91.3	6252	82.7	11067
	inuse	177	48.6	5484	83.0	2198	86.4	1753	84.2	1820	77.4	2726	92.1	1371	63.8	4044
	symm	56	48.2	1744	66.1	1642	50.0	1710	66.1	1188	19.6	2706	48.2	1812	48.2	1772
	norm	41	63.4	902	63.4	921	63.4	903	63.4	932	9.8	2222	63.4	915	58.5	1025
$\mathcal{T}_{\text{eval}}$	*array*	28	100.0	2	100.0	17	100.0	2	100.0	40	100.0	20	100.0	15	100.0	14
	init	790	88.9	5329	91.8	4426	91.8	4062	92.4	4293	84.8	8060	90.1	5332	80.2	10638
	inuse	172	75.6	2552	82.0	2158	86.6	1688	84.3	1935	62.2	4019	91.9	1014	64.5	3773
	symm	56	48.2	1800	66.1	1462	51.8	1625	66.1	1191	26.8	2473	51.8	1657	48.2	1781
	norm	30	50.0	901	50.0	909	50.0	901	50.0	910	50.0	924	50.0	913	50.0	902
$\mathcal{T}_{\text{array}}$	*array*	28	100.0	2	100.0	19	100.0	2	100.0	40	100.0	20	100.0	15	100.0	14
	init	582	99.3	290	100.0	450	99.0	553	100.0	381	95.7	2300	99.1	783	99.8	1241
	inuse	47	83.0	483	97.9	214	89.4	304	91.5	303	85.1	492	100.0	8	95.7	284
	symm	1337	99.2	737	100.0	1140	99.0	880	100.0	218	98.2	1613	99.4	704	99.1	835
	norm	32	59.4	782	59.4	789	59.4	782	56.2	843	59.4	783	59.4	818	59.4	886
$\mathcal{T}_{\text{array}^*}$	*array*	1457	99.9	156	99.9	957	99.9	130	99.9	356	99.5	775	99.9	166	99.9	215
	init	3825	99.2	2427	99.7	8005	99.3	2543	99.7	2022	95.0	13401	99.5	5065	99.7	4471
	inuse	3089	99.6	847	99.8	1112	99.7	610	99.7	717	99.5	1420	99.8	605	99.7	819
	symm	6403	99.8	1206	99.9	7333	99.6	1628	99.8	1181	83.5	64103	99.8	2362	99.6	2583
	norm	157	91.7	785	91.7	851	91.7	786	91.1	901	76.4	2246	91.7	961	90.4	991
$\mathcal{T}_{\text{policy}}$	*array*	26	100.0	2	100.0	17	100.0	2	100.0	36	100.0	19	100.0	14	100.0	14
	init	582	99.3	290	100.0	376	99.0	562	100.0	342	95.7	1826	99.1	1118	99.8	1531
	inuse	20	60.0	482	100.0	194	75.0	304	80.0	296	65.0	489	100.0	81	95.0	229
	symm	1345	99.2	685	100.0	1098	99.0	868	100.0	184	99.2	797	99.4	718	99.1	881
	norm	14	100.0	2	100.0	10	100.0	1	92.8	96	100.0	3	100.0	41	100.0	109

ably large number of obligations to the user. The only exception is the *array*-policy, which produces by far the simplest safety obligations. This confirms our second hypothesis: aggressive preprocessing is absolutely necessary to yield reasonable results.

Looking more closely at the different simplification stages, we can see that breaking the large original formulas into a large number of smaller but independent proof tasks ($\mathcal{T}_{\forall,\Rightarrow}$) boosts the relative performance considerably. However, due to the large absolute number of tasks, the absolute number of failed tasks (and thus the total response time) also increases. With each additional simplification step, the percentage of solved proof obligations increases further. Interestingly, however, $\mathcal{T}_{\forall,\Rightarrow}$ and $\mathcal{T}_{\text{array}}$ seem to have the biggest impact on performance. The reason seems to be that equality reasoning on deeply nested terms and

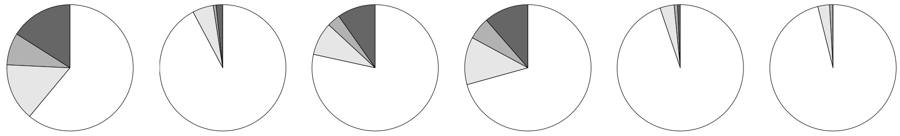

Fig. 4. Distribution of easy ($T_{proof} < 1s$, white), medium ($T_{proof} < 10s$, light gray), difficult ($T_{proof} < 60s$, gray) proofs, and failing proof tasks (dark gray) for the different simplification stages (prover: E-Setheo-03F). N is the total number of proof tasks at each stage.

formula structures can then be avoided, albeit at the cost of the substantial increase in the number of proof tasks. The results with the simplification strategy $\mathcal{T}_{\text{array}^*}$, which only contains the language-specific rules, also illustrates this behavior. The *norm*-policy clearly produces the most difficult safety obligations, requiring essentially inductive and higher-order reasoning. Here, all simplification steps are required to make the obligations go through the first-order ATPs.

Since our proof tasks are generated directly by a real application and are not hand-picked for certain properties, many of them are (almost) trivial—even in the unsimplified case. Figure 4 shows the resources required for the proof tasks as a series of pie charts for the different simplification stages. All numbers are obtained with E-Setheo-03F; the figures for the other provers look similar. Overall, the charts reflect the expected behavior: with additional preprocessing and simplification of the proof obligations, the number of easy proofs increases substantially and the number of failing proof tasks decreases sharply from approximately 16% to zero. The relative decrease of easy proofs from $\mathcal{T}_{\forall,\Rightarrow}$ to $\mathcal{T}_{\text{prop}}$ and $\mathcal{T}_{\text{eval}}$ is a consequence of the large number of easy proof tasks already discharged by the respective simplifications.

4.3. *Comparing Prover Variants*

The results in Table 4 indicate there is no single best theorem prover. Even variants of the "same" prover can differ widely in their results. Tables 5 and 6 thus compare the results within the same "prover families" in more detail.

For some proof obligations, the choice of the clausification module makes a big difference. The TPTP-converter implements a straightforward algorithm similar to the one described in (Ref. 41). Flotter[35,36] uses a highly elaborate conversion algorithm which performs many simplifications and avoids exponential increase in the number of generated clauses. This effect is most visible on the unsimplified obligations (e.g., $\mathcal{T}_{\emptyset}$ under *init*), where variants using the Flotter clausifier perform substantially better than the other variants.

A second, somewhat surprising observation is that the most recent prover variants do not improve over previous variants, and in many cases even regress. This is particularly pronounced for Vampire, where the most recent version Vampire 7.0 is almost consistently outperformed by the previous version Vampire 6.0 and barely manages to match the results of the even older version Vampire 5.0, both in terms of solved proof tasks and overall

Table 5. Detailed results for E-Setheo variants: Percentage of solved proof tasks and total ATP-times.

			03F		03n		04		04F		04T		Setheo	
$\mathcal{T}$	Policy	N	%	T_{ATP}	%	T_{ATP}	%	T_{ATP}	%	T_{ATP}	%	T_{ATP}	%	T_{ATP}
$\mathcal{T}_{\emptyset}$	*array*	110	99.1	180	97.3	240	98.2	150	96.4	330	95.5	390	98.2	141
	init	164	71.3	3173	7.9	9068	12.2	8699	55.5	4502	16.5	8274	9.1	8945
	inuse	19	57.9	571	47.4	660	47.4	607	63.1	443	47.4	610	52.6	605
	symm	18	55.5	551	16.7	902	16.7	900	55.5	500	16.7	902	16.7	901
	norm	54	68.5	1110	61.1	1281	61.1	1271	64.8	1172	61.1	1285	61.1	1268
$\mathcal{T}_{\forall,\Rightarrow}$	*array*	1457	99.9	1195	99.9	960	99.9	586	99.9	848	99.9	825	99.9	465
	init	3177	98.1	6872	97.9	10465	92.1	16395	92.1	17046	91.9	16065	97.6	6149
	inuse	1123	97.9	2647	97.2	2823	92.4	5510	91.7	6617	92.5	5438	97.6	1904
	symm	286	93.3	4012	93.3	4186	90.5	1800	90.5	2008	89.9	1930	93.3	1321
	norm	155	90.3	1092	90.3	983	90.3	944	90.3	1016	90.3	1008	90.3	934
$\mathcal{T}_{\text{prop}}$	*array*	275	99.3	257	99.3	262	99.3	230	99.3	388	99.3	371	99.3	189
	init	919	93.4	5042	92.7	5499	75.1	14147	74.5	14836	73.8	15177	91.6	5188
	inuse	177	86.4	1823	83.0	2198	52.0	5163	53.1	5028	52.5	5137	84.7	1777
	symm	56	66.1	1629	66.1	1642	51.8	1645	51.8	1666	48.2	1771	66.1	1170
	norm	41	63.4	903	63.4	921	63.4	913	63.4	924	63.4	922	63.4	906
$\mathcal{T}_{\text{eval}}$	*array*	28	100.0	24	100.0	17	100.0	15	100.0	29	100.0	10	100.0	8
	init	790	92.4	4350	91.8	4426	72.4	13441	70.4	14690	70.0	14821	90.4	4903
	inuse	172	85.5	1777	82.0	2158	51.2	5108	51.7	5108	51.2	5151	84.3	1715
	symm	56	66.1	1464	66.1	1462	51.8	1653	48.2	1796	48.2	1791	66.1	1188
	norm	30	50.0	912	50.0	909	50.0	908	50.0	914	50.0	914	50.0	904
$\mathcal{T}_{\text{array}}$	*array*	28	100.0	27	100.0	19	100.0	15	100.0	24	100.0	27	100.0	2
	init	582	100.0	575	100.0	450	99.8	338	99.1	983	97.4	1490	100.0	210
	inuse	47	93.6	234	97.9	214	95.7	145	95.7	168	95.7	163	100.0	17
	symm	1337	100.0	1595	100.0	1140	99.4	884	99.2	1755	99.2	1650	100.0	236
	norm	32	59.4	793	59.4	789	59.4	788	59.4	795	59.4	794	59.4	783
$\mathcal{T}_{\text{array}^*}$	*array*	1457	99.9	1275	99.9	957	99.9	390	99.9	1340	99.9	1167	99.9	444
	init	3825	99.7	5910	99.7	8005	99.3	2645	99.7	4840	99.4	4308	99.7	1664
	inuse	3089	99.8	2989	99.8	1112	99.7	1032	99.8	2894	99.8	2601	99.8	818
	symm	6403	99.9	4745	99.9	7333	99.8	1626	99.8	6567	99.8	5729	99.9	1587
	norm	157	91.7	876	91.7	851	91.7	821	91.7	878	91.7	789	91.7	810
$\mathcal{T}_{\text{policy}}$	*array*	26	100.0	19	100.0	17	100.0	13	100.0	26	100.0	4	100.0	7
	init	582	100.0	567	100.0	376	99.8	409	99.1	886	97.4	1495	100.0	254
	inuse	20	100.0	204	100.0	194	90.0	138	90.0	147	90.0	143	100.0	1
	symm	1345	100.0	1429	100.0	1098	99.4	870	99.2	1747	99.2	1635	100.0	348
	norm	14	100.0	1	100.0	10	100.0	9	100.0	15	100.0	14	100.0	4

proof times. A similar pattern also holds in the case of E-Setheo, where the most recent version E-Setheo-04 is generally outperformed by both the earlier development snapshot E-Setheo-03n and the E-free variant Setheo.[a]

[a]The slight advantage of E-Setheo-04 on the unsimplified proof tasks is a consequence of the change in the built-in clausification module; E-Setheo-03n still relies on the TPTP-clausifier which fails on a number of tasks while E-Setheo-04 uses the same clausifier as E0.82.

Table 6. Detailed results for Vampire and Gandalf variants: Percentage of solved proof tasks and total ATP-times.

$\mathcal{T}$	Policy	N	Vampire 7.0 %	Vampire 7.0 T_{ATP}	Vampire 6.0 %	Vampire 6.0 T_{ATP}	Vampire 5.0 %	Vampire 5.0 T_{ATP}	Gandalf %	Gandalf T_{ATP}	Gandalf-F %	Gandalf-F T_{ATP}
$\mathcal{T}_{\emptyset}$	*array*	110	98.2	123	98.2	170	98.2	122	97.3	215	98.2	140
	init	164	8.5	9004	8.5	9004	8.5	9003	23.2	8167	78.0	2712
	inuse	19	47.4	607	57.9	603	47.4	610	68.4	541	78.9	352
	symm	18	16.7	900	16.7	900	16.7	900	16.7	900	61.1	464
	norm	54	64.8	1146	61.1	1264	64.8	1163	68.5	1082	72.2	919
$\mathcal{T}_{\forall,\Rightarrow}$	*array*	1457	99.9	148	99.9	206	99.9	138	99.9	496	99.9	328
	init	3177	94.3	12449	94.4	15639	93.5	14421	98.1	4722	98.0	6283
	inuse	1123	93.8	4451	94.3	4531	94.2	4013	97.5	2063	98.6	1558
	symm	286	89.5	1919	87.8	2968	87.8	2467	90.5	1710	92.7	1384
	norm	155	89.0	1026	89.0	1030	89.0	1020	90.3	904	90.3	965
$\mathcal{T}_{\text{prop}}$	*array*	275	99.3	141	99.3	183	99.3	128	99.3	301	99.3	182
	init	919	82.6	10585	82.7	11067	81.8	11122	93.2	4718	93.0	4722
	inuse	177	61.0	4206	63.8	4044	64.4	3885	84.2	1820	91.0	1454
	symm	56	48.2	1750	48.2	1772	48.2	1758	66.1	1188	60.7	1353
	norm	41	58.5	1024	58.5	1025	58.5	1021	63.4	932	63.4	933
$\mathcal{T}_{\text{eval}}$	*array*	28	100.0	6	100.0	14	100.0	2	100.0	40	100.0	28
	init	790	79.9	10357	80.2	10638	80.0	9875	92.4	4293	91.8	4753
	inuse	172	59.3	4322	64.5	3773	65.7	3647	84.3	1935	90.7	1458
	symm	56	48.2	1751	48.2	1781	48.2	1742	66.1	1191	58.9	1416
	norm	30	43.3	1023	50.0	902	43.3	1020	50.0	910	50.0	905
$\mathcal{T}_{\text{array}}$	*array*	28	100.0	7	100.0	14	100.0	2	100.0	40	100.0	27
	init	582	99.1	1528	99.8	1241	99.0	902	100.0	381	100.0	532
	inuse	47	89.4	371	95.7	284	87.2	364	91.5	303	93.6	271
	symm	1337	98.8	1017	99.1	835	99.0	900	100.0	218	99.7	363
	norm	32	46.9	1024	59.4	886	46.9	1020	56.2	843	59.4	789
$\mathcal{T}_{\text{array}^*}$	*array*	1457	99.9	136	99.9	215	99.9	123	99.9	356	99.9	215
	init	3825	99.3	3761	99.7	4471	98.6	5674	99.7	2022	99.7	2937
	inuse	3089	99.7	656	99.7	819	99.6	777	99.7	717	99.7	618
	symm	6403	99.6	2028	99.6	2583	99.6	2071	99.8	1181	99.9	829
	norm	157	89.2	1026	90.4	991	89.2	1021	91.1	901	91.7	850
$\mathcal{T}_{\text{policy}}$	*array*	26	100.0	6	100.0	14	100.0	2	100.0	36	100.0	23
	init	582	99.1	1116	99.8	1531	99.0	903	100.0	342	100.0	724
	inuse	20	75.0	368	95.0	229	70.0	397	80.0	296	85.0	266
	symm	1345	98.8	1017	99.1	881	99.0	902	100.0	184	99.7	364
	norm	14	71.4	244	100.0	109	71.4	240	92.8	96	100.0	12

4.4. *Difficult Proof Tasks*

Since all proof tasks are generated in a uniform manner through the application of a safety policy by the VCG, it is expected that many of the difficult proof tasks share some structural similarities. Most safety obligations generated by the VCG are of the form $\mathcal{A} \Rightarrow \mathcal{B}_1 \wedge \ldots \wedge \mathcal{B}_n$ where the $\mathcal{B}_i$ are variable disjoint. These obligations can be split up into n smaller proof obligations of the form $\mathcal{A} \Rightarrow \mathcal{B}_i$ and most theorem provers can then handle these smaller independent obligations much more easily than the large original.

We have identified two other classes of hard examples; these classes are directly addressed by the rewrite rules of the policy-specific simplifications. The first class contains formulas of the form $symm(r) \Rightarrow symm(diag\text{-}updates(r))$. Here, r is a matrix variable which is updated along its diagonal, and we need to show that r remains symmetric after the updates. For a 2x2 matrix and two updates (i.e., $r_{00} = x$ and $r_{11} = y$), we obtain the following simplified version of an actual proof task:

$$\begin{aligned}
&\forall i, j \cdot (0 \leq i, j \leq 1 \Rightarrow sel(r, i, j) = sel(r, j, i)) \Rightarrow \\
&\quad (\forall k, l \cdot (0 \leq k, l \leq 1 \Rightarrow \\
&\qquad sel(upd(upd(r, 1, 1, y), 0, 0, x), k, l) = sel(upd(upd(r, 1, 1, y), 0, 0, x), l, k))).
\end{aligned}$$

This already pushes the provers to their limits—E-Setheo cannot prove this while Spass succeeds here but fails if the dimensions are increased to 3x3, or if three updates are made. In our examples, matrix dimensions up to 6x6 with 36 updates occur, yielding large proof obligations of this specific form which are not provable by current ATPs without further preprocessing.

Another class of seemingly trivial but hard examples, which frequently shows up in the *init*-policy, results from the expansion of deeply nested sel/upd-terms. These problems have the form

$$\forall i, j \cdot 0 \leq i \leq n \wedge 0 \leq j \leq n \Rightarrow (\neg(i = 0 \wedge j = 0) \wedge \ldots \wedge \neg(i = n \wedge j = n) \Rightarrow false)$$

and soon become intractable for the clausifier, even for small n ($n = 2$ or $n = 3$), although the proof would be easy after a successful clausification.

4.5. *Policy-Specific Domain Theories*

The domain theory described in Section 3.3 and used in the experiments summarized in Tables 4–6 contains *all* axioms required to prove *any* of the obligations; in particular, it also contains axioms which are specific to the symbols used only in one policy and which should thus not be required for any obligation from the other policies. However, experience shows that the ATPs have problems detecting such redundant axioms.[23,42,31]

In order to evaluate the effect of redundant axioms in our case, we used a reduced domain theory for the *array*, *init*, and *inuse* safety policies and then re-ran E-Setheo-04 and E. Here we used E-Setheo-04 because previous experiments with E-Setheo-03F have shown[6] that its scheduling algorithm is much more sensitive to modifications than that of E-Setheo-04. The reduced domain theory uses the same dynamic axiom generator as the full theory but omits seven axioms that specify the behavior of matrix operations (i.e., addition, subtraction, multiplication, transposition, and inversion) which do not occur in the obligations resulting from the above policies. The reduced set thus contains 37 axioms and 17 symbols.

Table 7 summarizes this experiment and gives the results and times for both the original full and the reduced domain theory. Note that T_{proof} only includes the CPU times for successful proof attempts; T_{mean} is the average CPU time for these cases. However, as common for such comparison experiments, the results are unfortunately not very conclusive. For E-Setheo-04, the effects are fairly small; the reduced domain theory leads to the "loss" of a single proof but allows an additional proof to be found in three versions of the *inuse*-tasks. With a few exceptions, the average proof times decrease by about 25% but this can to some

Table 7. Proof results and times—policy-specific domain theories.

			E-Setheo-04						E					
			Full theory			Reduced theory			Full theory			Reduced theory		
	Policy	N	%	T_{proof}	T_{mean}	%	T_{proof}	T_{mean}	%	T_{proof}	T_{mean}	%	T_{proof}	T_{mean}
$\mathcal{T}_{\emptyset}$	*array*	110	98.2	29.5	0.27	98.2	22.6	0.20	98.2	5.2	0.04	98.2	4.6	0.04
	init	164	12.2	59.2	2.96	12.2	50.7	2.53	20.7	60.3	1.77	22.6	175.2	4.73
	inuse	19	47.4	6.9	0.76	47.4	0.7	0.07	52.6	71.8	7.18	47.4	2.6	0.28
$\mathcal{T}_{\forall,\Rightarrow}$	*array*	1457	99.9	465.6	0.32	99.9	349.5	0.24	99.9	11.5	0.00	99.9	9.1	0.00
	init	3177	92.1	1395.4	0.47	92.1	1087.1	0.37	97.8	952.8	0.30	97.3	371.3	0.12
	inuse	1123	92.4	409.8	0.39	92.5	303.5	0.29	97.9	341.5	0.31	98.5	62.6	0.05
$\mathcal{T}_{\text{prop}}$	*array*	275	99.3	110.3	0.40	99.3	93.1	0.34	99.3	5.9	0.02	99.3	4.1	0.01
	init	919	75.1	406.7	0.58	75.1	185.9	0.26	92.4	248.2	0.29	90.6	261.9	0.31
	inuse	177	52.0	63.3	0.68	52.5	29.6	0.31	86.4	313.0	2.04	90.4	16.3	0.10
$\mathcal{T}_{\text{eval}}$	*array*	28	100.0	14.8	0.52	100.0	10.8	0.38	100.0	2.4	0.08	100.0	2.4	0.08
	init	790	72.4	361.0	0.63	72.3	167.5	0.29	91.8	161.9	0.22	89.2	742.8	1.05
	inuse	172	51.2	68.5	0.77	51.7	37.6	0.42	86.6	308.0	2.06	90.1	43.5	0.28
$\mathcal{T}_{\text{array}}$	*array*	28	100.0	15.1	0.53	100.0	11.1	0.39	100.0	2.5	0.08	100.0	2.2	0.07
	init	582	99.8	277.5	0.47	99.8	270.2	0.46	99.0	192.6	0.33	100.0	30.7	0.05
	inuse	47	95.7	24.8	0.55	95.7	6.9	0.15	89.4	4.5	0.10	93.6	5.0	0.11
$\mathcal{T}_{\text{array*}}$	*array*	1457	99.9	270.1	0.18	99.9	351.0	0.24	99.9	10.3	0.00	99.9	10.5	0.00
	init	3825	99.3	1084.9	0.28	99.3	1201.9	0.31	99.3	1043.2	0.27	99.5	221.5	0.05
	inuse	3089	99.7	552.5	0.17	99.7	451.8	0.14	99.7	10.2	0.00	99.7	27.3	0.00
$\mathcal{T}_{\text{policy}}$	*array*	26	100.0	13.0	0.50	100.0	9.9	0.38	100.0	2.3	0.08	100.0	1.8	0.06
	init	582	99.8	348.7	0.60	99.8	264.0	0.45	99.0	202.2	0.35	100.0	30.4	0.05
	inuse	20	90.0	18.3	1.01	90.0	13.9	0.77	75.0	4.2	0.28	85.0	4.7	0.27

extent already be explained by the fact that the preprocessing steps (mainly clausification) run faster on the smaller problems. For E, the effects are much more pronounced, but also much less uniform. In some cases, in particular for the unsimplified or only slightly simplified variants, E can find a substantial number of additional proofs, The average proof times are usually slightly better for the reduced axiom set but they can vary widely—up to one order of magnitude in both directions.

5. Proof Checking

For certification purposes, assurance or, better yet, explicit evidence must be provided that none of the individual tool components can yield incorrect results. The VCG is designed so that it can be manually inspected for correctness and, similarly, the rewrite rules used for simplification can be inspected and even individually proven correct. However, the state-of-the-art high-performance ATPs in our system use complicated calculi, elaborate data structures, and optimized implementations to increase their deductive power and to provide results quickly. This makes a formal verification of their correctness impossible in practice. Although they have been extensively validated by the theorem proving community (using the TPTP benchmark library), the ATPs remain the weakest link in the certification chain.

As an alternative to formally verifying the ATPs, they can be extended to generate sufficiently detailed proofs which can then be independently checked by a small and thus verifiable algorithm. This is the same approach we have taken in extending the synthesis system to generate annotated code, rather than directly verifying the system itself. However, although this idea is very simple in theory, there are currently (as of 2005) almost no dedicated proof checkers for high-performance ATPs.[b] This has a number of practical reasons. (i) Not all ATPs generate the detailed proofs which are required, mainly due to implementation effort and the run-time overheads this incurs. (ii) On-going changes in the ATP require frequent updates and re-verification of the proof checker. (iii) Almost all ATPs (in particular all the provers used here) work on problems in clausal normal form, so the proof checking can only be done on the clausal level, and not on the original formula level. Since clausification is an important step in the proof process, this reduces the confidence that proof checking can bring. (iv) Most ATPs contain a large number of high-level inference rules (such as *splitting*) which cannot easily be expanded into sequences of low-level inferences, making a proof checker more complicated and thus hard to verify.

In our experiments, we used the Ivy system,[28] which combines a clausifier and the Otter theorem prover with a proof checker. Ivy is implemented within the ACL2 logic,[43] and both the clausifier and the proof checker have been formally verified in ACL2. Ivy thus provides the same functionality as a verified theorem prover for first-order logic, but unlike a "correct-by-inspection" prover such as leanTAP[44] it generally achieves relatively good performance by using Otter to find the proofs. The formal verification of the Ivy clausifier and proof checker are based on finite domains[28] but since the implementation of Ivy does not actually rely on the finiteness, the system can be used for arbitrary proof tasks.

A more serious practical limitation of Ivy is caused by the implementation of the clausification algorithm, which only returns an unstructured clause set but does not maintain traceability between the clauses (or literals) and the positions they had in the original formula. This has a negative influence on the prover's behavior. Like many ATPs, Otter can be sped up considerably if it is known which parts of the formula are axioms and which belong to the original conjecture. This distinction allows the prover to apply goal-oriented rules. Our application naturally provides this information, but this is ignored by Ivy. Thus, Ivy can only use Otter's auto-mode which is not very well suited for our proof obligations. Experiments also revealed that Ivy has problems in handling the full axiom set. With the policy-specific domain theory described in Section 4.5, we obtained the following results for the fully simplified tasks: 100% in 34.8s for the *array* property, 89.2% in 4929.2s for *init*, and 65.0% in 657.5s for *inuse*.

6. Traceability

The successful application of an automated theorem prover to verification and, in particular, to certification problems such as we have described here, places more requirements on the prover than just raw deductive power. Since the aim of certification is to provide explicit evidence that software meets a specified standard of safety, it is important that domain experts can assess the evidence for successful checks of the safety properties and

[b]Many ATPs provide a weak form of proof checking, in which the prover itself (or even another ATP) is used to prove that the conclusion of each individual inference step logically follows from its premises. However, this only provides limited assurance because the checker ATP is not formally verified.

any places where they are violated. This traceability is also mandated by standards such as DO-178B.[45]

In practice, safety checks are generally carried out during code reviews,[46] where reviewers look in detail at each line of the code and check the individual safety properties statement by statement. The successful outcome of a code review, therefore, consists of the marked-up code, where each statement is labeled either with "complies with safety property P", or with information about the violation. Automating the link between code and certificates requires the system to generate two additional artifacts: (i) location information which links the safety obligations (or their proofs) to specific lines of code in the program being certified, and (ii) a summary which interprets the obligation in terms of the safety policy, program, and specification.

Source code locations are added by the VCG to the safety conditions and safety substitutions it constructs as it processes a statement with a given location. We currently use simple line numbers as locations and do not break them further down into individual subterm positions[47] because this finer level of detail it is not required for our purposes. However, the source locations need to be threaded through all stages of our certification architecture (cf. Figure 1), in particular the simplifier, in order to trace the resulting VCs back to their origins. Since each VC is generally linked to multiple statements, the location information for the entire program needs to be maintained, even if we just want to know whether a single line in the code satisfies a given safety property. We have thus extended the rewrite rules described in Section 3.2 to preserve the associated labels through the simplification process, similar to the labeled rewrite rules used in the Simplify prover.[48] This approach requires careful engineering to maintain the relevant location information while minimizing the scope of the labels and thus preventing the introduction of too much noise into the linking process.

We have also implemented a *certification browser* that displays the links between the verification conditions and the lines of the (annotated) code. This is especially useful when a VC cannot be proven, and it is not clear whether the error is due to a genuine safety violation in the program, an inadequate or erroneous annotation, an incorrect propagation, or a weakness in the domain theory or the prover.

Figure 5 shows a screen-shot of the browser which displays the program in the left frame, and the list of VCs (including their status and a link to the actual proof task) in the right frame. Linking works in both directions: clicking on a statement or annotation displays all VCs to which it contributes (i.e., which are labeled with its line number), and clicking on a VC highlights all statements and annotations that are affected by it. This two-way linking allows users not only to review the program line by line but also to quickly identify all potential error sources associated with an unproven VC.

In general, useful information extracted from an ATP can be used for purposes of auto-generating documentation. In (Ref. 49), we describe a *safety documentation* tool, which generates a natural language description explaining the safety of a program, by converting the VCs into text. We are currently integrating this with the certification browser, which requires the labels to be extended with other semantic information.

These tools can also be extended by carrying out some symbolic evaluation from the

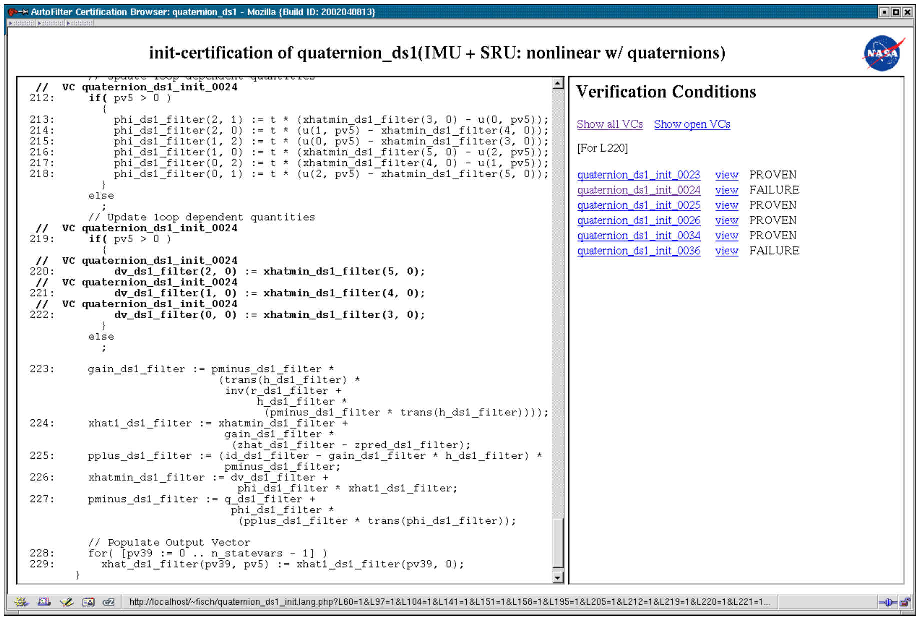

Fig. 5. Screen-shot of certification browser linking program and VCs.

simplifier as an intermediate step to using the full proofs. More detailed information (such as which axioms have been used) could be obtained by threading the tracing through the ATP itself and into the proof it generates. For example, the analysis needs to reveal which annotations are actually required to satisfy a property. The extraction of this information requires knowledge of which parts of the formula contributed to the proof, as well as their location information. The tracing problem is again aggravated by the fact that most theorem provers work on clausal normal form, which usually loses the important location information.

7. Conclusions

We have described a system for the automated certification of safety properties of NASA state estimation and data analysis software. The system uses a generic VCG together with explicit safety policies to generate policy-specific safety obligations which are then automatically processed by a first-order ATP. We have evaluated several state-of-the-art ATPs on more than 25,000 proof tasks generated by our system. We believe this to be the first comprehensive set of experiments looking at the suitability of a range of ATPs for software verification.

With "out-of-the-box" provers, only about two-thirds of the tasks could be proven but after aggressive simplification, most of the provers could solve almost all emerging tasks. In order to see the effects of simplification more clearly, we experimented with several preprocessing stages. Figure 6 shows (on the left) the overall results for the different stages and provers.

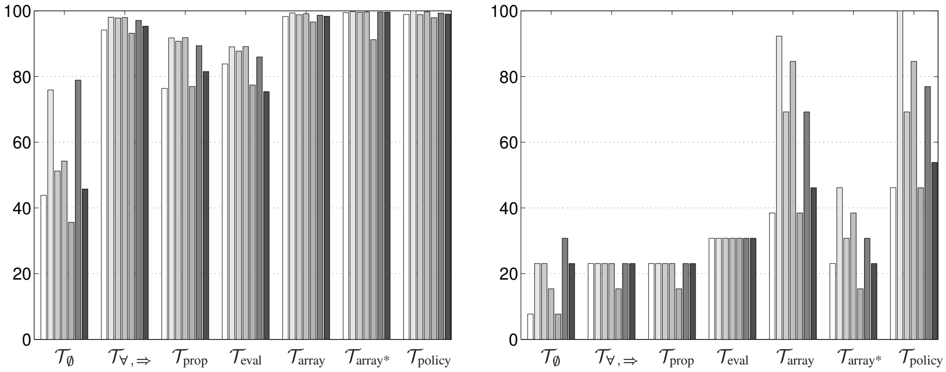

Fig. 6. Comparison of proof results (left) and certification results (right) for E-Setheo-03F, E, Gandalf, Otter, Spass, and Vampire 6.0 (left to right).

However, the percentage of solved *proof* tasks is a very ATP-centric metric; it is also somewhat artificial because it can easily be boosted by splitting the original obligations into a larger number of small proof tasks (cf. the results for $\mathcal{T}_\emptyset$ and $\mathcal{T}_{\forall,\Rightarrow}$). An empirically more meaningful metric for the success of this ATP-application is the percentage of solved *certification* tasks, i.e., the relative number of cases in which the ATP solves *all* safety obligations resulting from the application of a safety policy to an individual program. Figure 6 shows this metric (on the right) for the different simplification stages and provers. This change in perspective leads to a dramatic change in the interpretation of the same results. The two major differences are: (i) the numbers go down and (ii) the variation between the provers becomes smaller for the unsimplified versions and substantially larger for the simplified versions. Both differences result from a few hard proof tasks which are distributed evenly over the different certification tasks. Consequently, empirical success is a lot harder to come by if it is defined in terms of the application rather than in terms of the TPTP corpus. However, as our experiments show it is clearly not impossible.

It is well-known that, in contrast to traditional mathematics, software verification requires large numbers of mathematically shallow (in terms of the concepts involved) but structurally complex proof tasks, yet current provers are not well suited to this. Since the propositional structure of a formula is of great importance, we believe that clausification algorithms should integrate more simplification and split goal tasks into independent subtasks. Likewise, certain application-specific constructs (e.g., *sel/upd*) can easily lead to proof tasks which cannot be handled by current ATPs. The reason is that simple manipulations on deeply nested terms, when combined with equational reasoning, can result in a huge search space.

Our certification approach relies on proof checking to ensure that the proofs are correct. However, the ATPs fare less well when assessed in these terms and more research efforts should go into the development of proof checkers for high-performance provers. Moreover, it is very difficult to get useful information from the ATPs which can then be used as a basis for documentation. Since we believe that software certification is potentially one of the main application areas for automated theorem proving, this is clearly another area in need of further work.

With our approach to certification of auto-generated code, we are able to automatically produce safety certificates for code of considerable size and structural complexity. By combining rewriting with state-of-the-art automated theorem proving, we obtain a safety certification tool which compares favorably with tools based on static analysis (see (Ref. 50) for a comparison).

Our current efforts focus on extending the certification system in a number of areas. One aim is to develop a certificate management system, along the lines of the Programatica project.[51] We are extending the interactive browser described in Section 6 to also link to proofs and design documents, and provide rich explanations of proof obligations, thus combining our work on certification with automated safety and design document generation.[49] In another thread of future work we plan to experiment with other reasoning systems and tools based on decision procedures (such as PVS and Simplify) to process our verification conditions. In this paper we have deliberately concentrated on using first-order theorem provers. Finally, we continue to integrate additional safety properties.

Acknowledgments

Bill McCune, Tanel Tammet, Stephan Schulz, Gernot Stenz, and Geoff Sutcliffe helped with the installation and integration of the different provers. Phil Oh helped with the evaluation scripts.

References

1. M. Whalen, J. Schumann and B. Fischer, AutoBayes/CC — Combining Program Synthesis with Automatic Code Certification (System Description), in *Proc. 18th Int. Conf. Automated Deduction*, Lect. Notes Artif. Intelligence 2392, ed. A. Voronkov (Springer, Berlin, 2002), pp. 290–294.
2. M. Whalen, J. Schumann and B. Fischer, Synthesizing Certified Code, in *Proc. Int. Symp. Formal Methods Europe 2002: Formal Methods—Getting IT Right*, Lect. Notes Comp. Sci. 2391, eds. L.-H. Eriksson and P. A. Lindsay (Springer, Berlin, 2002), pp. 431–450.
3. J. Whittle and J. Schumann, Automating the Implementation of Kalman Filter Algorithms. *ACM Transactions on Mathematical Software*, 2005. To appear.
4. B. Fischer and J. Schumann, AutoBayes: A System for Generating Data Analysis Programs from Statistical Models, *J. Functional Programming*, **13**(2003) 483–508.
5. E. Denney, B. Fischer and J. Schumann, An Empirical Evaluation of Automated Theorem Provers in Software Certification, in *Proc. IJCAR 2004 Workshop on Empirically Successful First Order Reasoning (ESFOR)*, 2004.
6. E. Denney, B. Fischer and J. Schumann, Using Automated Theorem Provers to Certify Auto-Generated Aerospace Software, in *Proc. Second Int. Joint Conf. Automated Reasoning*, Lect. Notes Artif. Intelligence 3097, eds. D. Basin and M. Rusinowitch (Springer, Berlin, 2004), pp. 198–212.
7. W. Reif, The KIV Approach to Software Verification, in *KORSO: Methods, Languages and Tools for the Construction of Correct Software*, Lect. Notes Comp. Sci. 1009, eds. M. Broy and S. Jähnichen (Springer, Berlin, 1995), pp. 339–370.
8. W. Reif, G. Schellhorn, K. Stenzel and M. Balser, Structured Specifications and Interactive Proofs with KIV, in (Ref. 52), pp. 13–40.
9. P. Homeier and D. Martin, Trustworthy Tools for Trustworthy Programs: A Verified Verification Condition Generator, in *Int. Workshop on Higher Order Logic Theorem Proving and its Applica-*

tions, Lect. Notes Comp. Sci. 859, eds. T.F. Melham and J. Camilleri (Springer, Berlin, 1994), pp. 269–284.

10. D. von Oheimb and T. Nipkow, Machine-checking the Java Specification: Proving Type-Safety, in *Formal Syntax and Semantics of Java*, Lect. Notes Comp. Sci. 1523, ed. J. Alves-Foss (Springer, Berlin, 1999), pp. 119–156.
11. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe and R. Stata, Extended static checking for Java, in *Proc. ACM Conf. Programming Language Design and Implementation 2002*, ed. L. J. Hendren (ACM Press, New York, 2002), pp. 234–245.
12. C. Flanagan and K. R. M. Leino, Houdini, an Annotation Assistant for ESC/Java, in *Proc. Int. Symp. Formal Methods Europe 2001: Formal Methods for Increasing Software Productivity*, Lect. Notes Comp. Sci. 2021, eds. J. Oliveira and P. Zave (Springer, Berlin, 2001), pp. 500–517.
13. B. Jacobs and E. Poll, Java program verification at Nijmegen: Developments and perspective, Technical Report NIII-R0318, Dept. of Computer Science, University of Nijmegen, 2003.
14. J. van den Berg and B. Jacobs, The LOOP Compiler for Java and JML, in *Proc. 7th Int. Workshop Tools and Algorithms for the Construction and Analysis of Systems*, Lect. Notes Comp. Sci. 2031, eds. T. Margaria and W. Yi (Springer, Berlin, 2001), pp. 299–312.
15. G. T. Leavens, A. L. Baker and C. Ruby, JML: A Notation for Detailed Design, in *Behavioral Specifications of Businesses and Systems* eds. H. Kilov, B. Rumpe and I. Simmonds (Kluwer, Dordrecht, 1999), pp. 175–188.
16. G. C. Necula, Proof-Carrying Code, in *Proc. 24th ACM Symp. Principles of Programming Languages*, (ACM Press, New York, 1997), pp. 106–19.
17. G. C. Necula and P. Lee, The Design and Implementation of a Certifying Compiler, in *Proc. ACM Conf. Programming Language Design and Implementation 1998*, ed. K. D. Cooper (ACM Press, New York, 1998), pp. 333–344.
18. A. W. Appel, N. Michael, A. Stump and R. Virga, A Trustworthy Proof Checker, *J. Automated Reasoning*, **31**(2003) 191–229.
19. G. Sutcliffe and C. Suttner, The CADE-19 ATP System Competition, *AI Communications*, **17**(2004) 103–110.
20. G. Sutcliffe, The CADE-J2 ATP System Competition, 2004. `www.tptp.org/CASC/J2/`.
21. G. Sutcliffe, C. B. Suttner and T. Yemenis, The TPTP Problem Library, in *Proc. 12th Int. Conf. Automated Deduction*, Lect. Notes Artif. Intelligence 814, ed. A. Bundy (Springer, Berlin, 1994), pp. 252–266.
22. G. Sutcliffe and C. Suttner, TPTP Home Page, 2003. `www.tptp.org`.
23. B. Fischer, *Deduction-Based Software Component Retrieval*, PhD thesis, U Passau, 2001. Available at `http://elib.ub.uni-passau.de/opus/volltexte/2002/23/`.
24. B. Fischer, J. Schumann and G. Snelting, Deduction-Based Software Component Retrieval, in (Ref. 52), pp. 265–292.
25. G. Brat and A. Venet, Precise and Scalable Static Program Analysis of NASA Flight Software, in *Proc. IEEE Aerospace Conf.*, IEEE Comp. Soc. Press, 2005. To appear.
26. G. J. Holzmann, *The Spin Model Checker: Primer and Reference Manual* (Addison-Wesley, 2003).
27. E. Denney and B. Fischer, Correctness of Source-Level Safety Policies, in *Proc. FM 2003: Formal Methods*, Lect. Notes Comp. Sci. 2805, eds. K. Araki, S. Gnesi and D. Mandrioli (Springer, Berlin, 2003), pp. 894–913.
28. W. McCune and O. Shumsky, System description: Ivy, in *Proc. 17th Int. Conf. Automated Deduction*, Lect. Notes Artif. Intelligence 1831, ed. D. McAllester (Springer, Berlin, 2000), pp. 401–405.
29. B. Fischer, A. Hajian, K. Knuth and J. Schumann, Automatic Derivation of Statistical Data Analysis Algorithms: Planetary Nebulae and Beyond, in *Proc. 23rd Int. Workshop on Bayesian Inference and Maximum Entropy Methods in Science and Engineering* eds. G. Erickson and Y. Zhai (American Institute of Physics, 2003), pp. 276–291.

30. B. Fischer and J. Schumann, Applying AutoBayes to the Analysis of Planetary Nebulae Images, in *Proc. 18th Int. Conf. Automated Software Engineering* eds. J. Grundy and J. Penix (IEEE Comp. Soc. Press, 2003), pp. 337–342.
31. J. Schumann, *Automated Theorem Proving in Software Engineering* (Springer, Berlin, 2001).
32. J. McCarthy, Towards a Mathematical Science of Computation, in *Proc. IFIP Congress 1962* (North-Holland, Amsterdam, 1962), pp. 21–28.
33. R. Letz and G. Stenz, DCTP: A Disconnection Calculus Theorem Prover, in *Proc. First Int. Joint Conf. Automated Reasoning*, Lect. Notes Artif. Intelligence 2083, eds. R. Gore, A. Leitsch and T. Nipkow (Springer, Berlin, 2001), pp. 381–385.
34. M. Moser, O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann and K. Mayr, The Model Elimination Provers SETHEO and E-SETHEO, *J. Automated Reasoning*, **18**(1997) 237–246.
35. C. Weidenbach, SPASS Home Page, 2003. `http://spass.mpi-sb.mpg.de`.
36. C. Weidenbach, B. Gaede and G. Rock, Spass and Flotter version 0.42, in *Proc. 13th Int. Conf. Automated Deduction*, Lect. Notes Artif. Intelligence 1104, eds. M. A. McRobbie and J. K. Slaney (Springer, Berlin, 1996), pp. 141–145.
37. S. Schulz, E — A Brainiac Theorem Prover, *AI Communications*, **15**(2002) 111–126.
38. T. Tammet, Gandalf, *J. Automated Reasoning*, **18**(1997) 199–204.
39. A. Riazanov and A. Voronkov, The Design and Implementation of Vampire, *AI Communications*, **15**(2002) 91–110.
40. W. McCune and L. Wos, Otter—The CADE-13 Competition Incarnations, *J. Automated Reasoning*, **18**(1997) 211–220.
41. D. W. Loveland, *Automated Theorem Proving: a Logical Basis* (North–Holland, Amsterdam, 1978).
42. W. Reif and G. Schellhorn, Theorem Proving in Large Theories. in (Ref. 52), pp. 225–242.
43. M. Kaufmann and J S. Moore, An Industrial Strength Theorem Prover for a Logic Based on Common Lisp, *Software Engineering*, **23**(1997) 203–213.
44. B. Beckert and J. Posegga, leanT^AP: Lean Tableau-based Deduction, *J. Automated Reasoning*, **15**(1995) 339–358.
45. RTCA Special Committee 167, Software Considerations in Airborne Systems and Equipment Certification, Technical report, RTCA, Inc., December 1992.
46. S. Nelson and J. Schumann, What makes a Code Review Trustworthy? in *Proc. Thirty-Seventh Annual Hawaii Int. Conf. on System Sciences (HICSS-37)*. IEEE Comp. Soc. Press, 2004.
47. R. Fraer, Tracing the Origins of Verification Conditions, in *Proc. 5th Int. Conf. on Algebraic Methodology and Software Technology*, Lect. Notes Comp. Sci. 1101, eds. M. Wirsing and M. Nivat (Springer, Berlin, 1996), pp. 241–255.
48. D. L. Detlefs, G. Nelson and J. B. Saxe. Simplify: A Theorem Prover for Program Checking, Technical Report HPL-2003-148, HP Labs, 2003,
49. E. Denney and R. P. Venkatesan, A Generic Software Safety Document Generator, in *Proc. 10th Int. Conf. on Algebraic Methodology and Software Technology*, Lect. Notes Comp. Sci. 3097, eds. C. Rattray, S. Maharaj and C. Shankland (Springer, Berlin, 2004), pp. 102–116.
50. E. Denney, B. Fischer and J. Schumann, Adding Assurance to Automatically Generated Code, in *Proc. 8th Int. Symp. High-Assurance Systems Engineering*, ed. C. V. Ramamoorthy (IEEE Comp. Soc. Press, 2004), pp. 297–299.
51. The Programatica Team, Programatica Tools for Certifiable, Auditable Development of High-assurance Systems in Haskell, in *Proc. High Confidence Software and Systems Conf.*, 2003. Available via `www.cse.ogi.edu/PacSoft/projects/programatica`.
52. W. Bibel and P. H. Schmitt (eds.), *Automated Deduction — A Basis for Applications* (Kluwer, Dordrecht, 1998).