An Empirical Evaluation of Automated Theorem Provers in Software Certification

Ewen Denney [1]

QSS / NASA Ames Research Center
Moffett Field, CA, USA

Bernd Fischer [2] Johann Schumann [3]

RIACS / NASA Ames Research Center
Moffett Field, CA, USA

Abstract

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, and usability. For full automation, however, 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 certification experiments that lead to 366 top-level safety obligations and ultimately to more than 25,000 proof tasks which have each been attempted by Vampire, Spass, e-setheo, and Otter. The proofs found by Otter have been proof-checked by IVY.

Key words: software certification, automated theorem proving, program synthesis, proof checking, 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

[1] Email: edenney@email.arc.nasa.gov
[2] Email: fisch@email.arc.nasa.gov
[3] Email: schumann@email.arc.nasa.gov

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

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 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 aim of this work is to increase trust in the code generator.

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 hinges on it. Second, the ability to generate proof objects and to carry out *proof checking* is essential to create explicit certificates. Third, there are a range of *traceability* issues which have 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. 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 aggressively simplify the obligations before they are handed over to the prover, taking advantage of domain-specific knowledge.

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 five 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 polices, 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 traceability issues. Finally, Section 7 draws some conclusions.

Conceptually, this paper continues the work described in [35,36] 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 [37] as well as with the AUTOBAYES system [12] 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. A shorter version of this paper appears as [5].

Related Work Our approach is related to proof-carrying code (PCC) [22]. 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 [23] could be used to ensure that the compilation step does not compromise the demonstrated safety policies. PCC also spawned an entire cottage industry of proof checkers, e.g., [1]; however, these use various higher-order logics and so are not applicable for our purposes.

Program verification is a popular application domain for theorem provers; we mention only a few systems here. KIV [26,28] is an interactive verification environment which can use different ATPs but relies heavily on term rewriting and user guidance. Sunrise [15] is a fully automatic system but uses custom-designed tactics in HOL to discharge the obligations. ESC/Java [10] is an automatic verification system but relies on the user to provide additional information on the program, e.g., loop invariants. Houdini [9] is an automatic annotation assistant which guesses invariants, but a significant amount of user interaction remains.

2 System Architecture

Our certification tool is built as an extension to the AUTOBAYES and AUTOFILTER 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

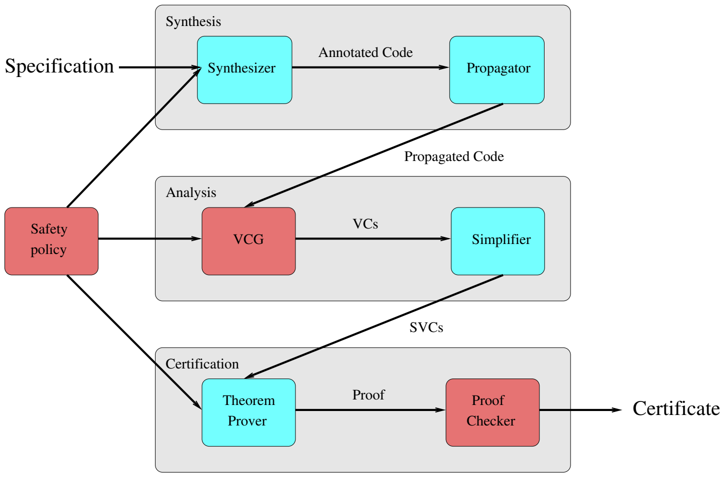

Fig. 1. Certification system architecture

an overview of the overall system architecture. 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 synthesizers 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, the *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 individual components are described in some detail in the subsequent sections. We distinguish *trusted* and *untrusted* components, shown in red (dark grey) and blue (light grey), respectively. In particular, the correctness of 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.

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

Table 1
Safety conditions for different policies

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 [3].

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 assigned a defined 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 (*in-use*) 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 [3].

From our perspective, 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 thus are 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. The fully annotated code is then processed by the VCG, which applies the rules of the safety policy to the annotated code in order to generate the safety conditions. As usual, the VCG works backwards through the code. At each line, the safety conditions are generated and the 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 [19].

The structure of a typical safety obligation (after substitution reduction and simplification) is given in Figure 3. It corresponds to the initialization safety of an assignment within a nested loop (given in Figure 2, including the generated invariants but omitting the postconditions). 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. A small script then adds the axioms of the domain theory, before the completed proof task is processed by the theorem

```
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

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

Fig. 3. Structure of a safety obligation

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 [32].

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 [37]. `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 of planetary nebula images taken by the Hubble Space Telescope (see [7,11] for more details). Although these data analysis applications are not safety-critical, they can run onboard a spacecraft thus making the software subject to qualification. `seg` describes an image segmentation problem for which an iterative (numerical) statistical clustering algorithm is synthesized. Finally, `gau` 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/ijcar.

3.2 Simplification

Proof task simplification is an important and integral part of our overall architecture. However, as observed before [13,8,30], 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 with 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 $x > y \rightsquigarrow \neg\, x \leq y$ 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 $n + 1 \rightsquigarrow succ(n)$ and $n - 1 \rightsquigarrow pred(n)$. 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.

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

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 [18], i.e., $\mathit{sel}(\mathit{upd}(a,i,v),j) \rightsquigarrow i = j \,?\, v : \mathit{sel}(a,j)$ 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., $\mathit{sel}(\mathit{constarray}(v,d),i) \rightsquigarrow v$.

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 language normalization step. $\mathcal{T}_{\text{array}}$ thus also contains two rules $\mathit{true} \,?\, x : y \rightsquigarrow x$ and $\mathit{false} \,?\, x : y \rightsquigarrow y$ 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

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

which is derived from the finite induction axiom to handle the result of simplifying nested *sel*/*upd*-terms. For *in-use*, we need a single rule $\mathit{def} = \mathit{use} \rightsquigarrow \mathit{false}$, 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

$$\begin{aligned} &\mathit{sel}(A + BCB^T, i, j) = \mathit{sel}(A + BCB^T, j, i) \\ &\rightsquigarrow \mathit{sel}(A, i, j) = \mathit{sel}(A, j, i) \wedge \mathit{sel}(C, i, j) = \mathit{sel}(C, j, i) \end{aligned}$$

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:

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

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:

$$P \wedge (\forall i \cdot 0 \le i \le n \Rightarrow x = \mathit{sel}(f,i)) \Rightarrow \sum_{i=0}^{n} \mathit{sel}(f,i) = 1$$
$$\rightsquigarrow P \wedge (\forall i \cdot 0 \le i \le n \Rightarrow x = \mathit{sel}(f,i)) \Rightarrow \sum_{i=0}^{n} x = 1$$

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 $P \,?\, T : F = R \rightsquigarrow (P \Rightarrow T = R) \wedge (\neg P \Rightarrow F = R)$ 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.

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 language normalization followed by the propositional simplifications $\mathcal{T}_{\text{prop}}$ and the finite induction rule. Similarly, $\mathcal{T}_{\text{array*}}$ is followed by the language 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 language normalization and clean-up.

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

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

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 because most theorem provers cannot calculate with integers, and to avoid the generation of large terms of the form $\mathit{succ}(\ldots\mathit{succ}(0)\ldots)$. For all integer literals n, m 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., $n = \mathit{succ}^n(0)$ and add a finite induction 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.

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 CASC-19 [31] prover competition in the FOL category. We used two versions of e-setheo [21] which were both derived from the CASC version. For e-setheo-csp03F, Flotter V2.1 [33,34] was used to convert the formulas into a set of clauses instead of the clausifier provided by the TPTP toolset [32]. e-setheo-new is a recent development version with several improvements over the original e-setheo-csp03 version. However, neither of the two versions of e-setheo was tuned in any way for this set of proof tasks. Both versions of Vampire [29] have been taken directly "out of the box"—they are the versions which were used at CASC-19. Spass 2.1 was obtained from the developer's website [33]. For comparison purposes, we also used Otter V3.2 [20], which has been essentially unchanged since 1996.

In the experiments, we used the default parameter settings and none of the special features of the provers. The only exception is Otter, where the developer provided an alternative parameter setting since the defaults proved unsuitable. 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.

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 of the generated programs (without annotations), the applicable safety policies, the size of the generated annotations (before propagation), and then, for each simplifier, the elapsed time T and the number N of generated obligations. 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

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

Table 3
Generation of safety obligations

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 *in-use*-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 change 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 the 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 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 ($\mathcal{T}_{\text{array}^*}$) only generally leads to an increase over $\mathcal{T}_{\forall,\Rightarrow}$ in the number of obligations; this even surpasses 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 *in-use* 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 invalid obligations from the *in-use*-policy), grouped by the different simplification levels. Each line in the table corresponds to the proof tasks originating from a specific safety policy (*array*, *init*, *in-use*, *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 ($\mathcal{T}_{\mathrm{policy}}$), all provers are able to find proofs for all tasks originating from at least one safety policy; e-setheo-csp03F 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 unacceptably 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.

Let us now look more closely at the different simplification stages. 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 also increases. With each additional simplification step, the percentage of solved proof obligations increases further. Interestingly, however, $\mathcal{T}_{\forall,\Rightarrow}$ and $\mathcal{T}_{\mathrm{array}}$ seem to have the biggest impact on performance. The reason seems to be that equality reasoning on deeply nested terms and 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 $\mathcal{T}_{\mathrm{array}^*}$, 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.

The results in Table 4 also indicate there is no single best theorem prover. Even variants of the "same" prover can differ widely in their results. 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 [17]. Flotter 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*),

		e-setheo03F		e-setheo-new		SPASS		Vampire6.0		Vampire5.0		Otter	
P	N	%	T_{ATP}	%	T_{ATP}	%	T_{ATP}	%	T_{ATP}	%	T_{ATP}	%	T_{ATP}
$\mathcal{T}_{\emptyset}$													
ar	110	96.4	192.4	94.5	284.9	96.4	73.4	95.5	178.1	95.5	102.1	83.6	870.3
in	164	76.8	3000.8	13.1	1759.8	75.0	2898.3	8.5	9224.9	8.5	8251.0	6.6	6534.7
iu	19	57.9	610.8	44.4	612.2	68.4	512.8	57.9	773.1	47.4	645.5	35.7	16.0
sy	18	50.0	387.7	8.3	266.1	38.9	555.3	16.7	744.9	16.7	723.6	16.7	847.9
no	54	51.9	1282.4	51.9	1341.0	51.9	1224.2	50.0	1316.5	48.1	1327.1	31.5	1537.7
$\mathcal{T}_{\forall,\Rightarrow}$													
ar	1457	99.0	903.4	94.2	5925.0	99.8	217.0	99.9	240.5	99.8	152.4	99.5	714.5
in	3177	88.4	3969.4	91.7	20784.8	97.4	8732.2	95.0	14482.2	93.5	14203.4	92.0	19310.5
iu	1123	59.3	819.1	96.4	4100.3	99.1	1733.5	95.3	4183.7	94.3	4206.8	96.6	3014.2
sy	286	93.4	1785.9	90.6	2341.0	88.5	3638.7	90.2	3315.8	91.3	1789.2	80.8	2160.1
no	155	85.8	1422.1	73.5	2552.5	84.5	1572.0	87.7	1359.9	87.1	1276.0	76.1	2051.6
$\mathcal{T}_{\text{prop}}$													
ar	275	99.3	278.2	76.4	4080.8	99.3	157.5	99.3	187.5	99.3	132.6	97.1	621.4
in	919	94.7	4239.4	73.0	17472.2	92.8	5469.7	84.9	10598.0	83.2	10546.8	78.8	13461.6
iu	177	86.4	1854.0	77.4	2768.2	94.9	1008.3	70.1	3806.2	65.0	3960.6	78.5	2729.2
sy	56	66.1	1476.2	51.8	1944.4	48.2	1911.3	58.9	1596.7	58.9	1424.8	19.6	2002.2
no	41	46.3	1361.2	41.5	1484.6	41.5	1478.2	53.7	1286.7	51.2	1275.3	9.8	2036.2
$\mathcal{T}_{\text{eval}}$													
ar	28	100.0	16.2	100.0	19.7	100.0	10.4	100.0	12.7	100.0	1.7	100.0	20.7
in	790	94.6	3944.2	94.1	8288.0	93.3	4380.1	82.5	10239.0	82.0	9040.2	85.7	7983.6
iu	172	86.0	1852.2	83.1	2305.2	94.8	1023.1	69.8	3718.1	67.4	3561.1	64.0	3715.3
sy	56	66.1	1451.1	66.1	1500.4	51.8	1716.0	62.5	1455.5	58.9	1389.8	26.8	1828.2
no	30	53.3	859.4	13.3	1575.8	50.0	940.5	66.7	736.7	53.3	858.0	50.0	1007.7
$\mathcal{T}_{\text{array}}$													
ar	28	100.0	15.4	100.0	19.8	100.0	10.4	100.0	12.7	100.0	1.7	100.0	20.2
in	582	100.0	527.6	100.0	823.9	99.7	875.8	100.0	1401.3	99.0	785.1	95.7	2468.7
iu	47	100.0	323.9	100.0	343.2	100.0	171.3	100.0	262.6	87.2	525.2	85.1	613.7
sy	1337	100.0	1104.3	99.9	1629.3	99.4	746.4	99.1	963.9	99.0	922.7	98.2	872.9
no	32	59.4	678.4	18.8	1583.1	59.4	709.7	62.5	791.7	50.0	858.6	59.4	896.2
$\mathcal{T}_{\text{array*}}$													
ar	1457	99.9	916.4	94.2	5918.0	99.9	210.8	99.9	240.6	99.9	153.1	99.5	711.9
in	3825	99.7	3412.3	96.3	13536.1	99.5	4574.9	99.8	4952.1	98.4	6000.1	95.5	13680.4
iu	3089	99.8	2438.4	99.4	5139.0	99.8	889.2	99.8	793.5	99.6	925.9	99.5	1427.8
sy	6403	99.9	5317.4	99.7	11787.7	99.7	3385.1	99.6	3277.3	99.6	1807.0	83.5	1682.8
no	157	86.0	1306.8	72.6	2670.8	86.0	1351.3	86.6	1449.9	86.0	1276.2	76.4	2078.3
$\mathcal{T}_{\text{policy}}$													
ar	26	100.0	15.0	100.0	17.7	100.0	9.9	100.0	12.0	100.0	1.6	100.0	19.7
in	582	100.0	529.2	100.0	827.9	99.5	875.2	100.0	1418.9	99.0	782.5	95.7	2456.7
iu	20	100.0	281.7	100.0	329.7	100.0	170.7	100.0	262.6	70.0	524.8	65.0	601.1
sy	1345	100.0	1104.6	99.9	1640.5	99.4	760.0	99.1	1048.8	99.0	926.9	99.3	501.1
no	14	100.0	9.0	57.1	375.8	100.0	26.2	100.0	108.0	71.4	241.8	100.0	69.7

Table 4
Results and times for *array* (*ar*), *init* (*in*), *in-use* (*iu*), *symm* (*sy*), and *norm* (*no*) policy.

where Spass and e-setheo-csp03F—which both use the Flotter clausifier—perform substantially better than the other provers.

Since our proof tasks are generated directly by a real application and are not

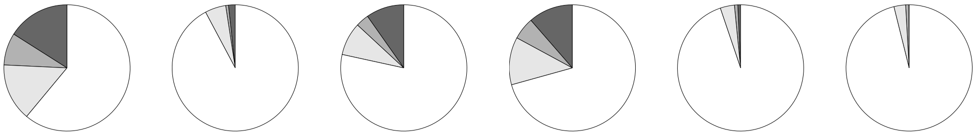

Fig. 4. Distribution of easy ($T_{proof} < 1s$, white), medium ($T_{proof} < 10s$, light grey), difficult ($T_{proof} < 60s$, dark grey) proofs, and failing proof tasks (black) for the different simplification stages (prover: e-setheo-csp03F). N is the total number of proof tasks at each stage.

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-csp03F; 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 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 obvious that many of the difficult proof tasks share some structural similarities. We have identified three classes of hard examples; these classes are directly addressed by the rewrite rules of the policy-specific simplifications.

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. The second class contains formulas of the form $\mathit{symm}(r) \Rightarrow \mathit{symm}(\mathit{diag\text{-}updates}(r))$. 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:

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

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

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

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

4.4 Policy-Specific Domain Theories

The domain theory described in Section 3.3 and used in the experiments summarized in Table 4 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 [8,27,30].

		e-setheo03F						Vampire 5.0					
		reduced theory			full theory			reduced theory			full theory		
simp.	policy	%	T_{proof}	T_{mean}	%	T_{proof}	T_{mean}	%	T_{proof}	T_{mean}	%	T_{proof}	T_{mean}
$\mathcal{T}_\emptyset$	*array*	96.4	61.5	0.56	96.4	131.2	1.19	96.6	0.4	0.01	95.5	3.1	0.03
	init	86.6	868.8	5.30	76.8	928.9	5.66	4.5	0.0	0.00	8.5	22.0	0.13
	in-use	57.9	32.0	1.68	57.9	62.4	3.28	47.4	7.4	0.4	47.4	7.5	0.40
$\mathcal{T}_{\forall,\Rightarrow}$	*array*	99.9	633.2	0.43	99.0	782.1	0.54	99.9	22.6	0.02	99.8	31.9	0.02
	init	98.6	7259.1	2.28	88.4	2155.1	0.68	93.5	1730.5	0.54	93.5	1845.2	0.58
	in-use	98.0	686.5	0.61	59.3	456.6	0.41	94.4	216.0	0.19	94.3	228.3	0.20
$\mathcal{T}_{\text{prop}}$	*array*	99.3	125.6	0.46	99.3	156.8	0.57	99.3	8.4	0.03	99.3	12.1	0.04
	init	95.2	5467.7	5.95	94.7	1274.4	1.39	83.0	1107.5	1.21	83.2	1258.5	1.37
	in-use	87.0	179.0	1.01	86.4	283.4	1.60	65.5	100.0	0.57	65.0	101.6	0.57
$\mathcal{T}_{\text{eval}}$	*array*	100.0	12.7	0.45	100.0	16.2	0.58	100.0	1.5	0.05	100.0	1.7	0.06
	init	94.7	5240.3	6.63	94.6	1342.1	1.70	82.3	491.4	0.62	82.0	478.3	0.61
	in-use	86.6	244.8	1.42	86.0	281.7	1.64	66.9	93.0	0.54	67.4	123.4	0.72
$\mathcal{T}_{\text{array}}$	*array*	100.0	12.4	0.44	100.0	15.4	0.55	100.0	1.4	0.05	100.0	1.7	0.06
	init	100.0	354.5	0.61	100.0	527.6	0.91	99.3	443.1	0.76	99.0	423.3	0.73
	in-use	100.0	31.4	0.67	100.0	203.4	4.33	87.2	39.5	0.84	87.2	42.9	0.91
$\mathcal{T}_{\text{array}^*}$	*array*	99.9	616.3	0.42	99.9	795.4	0.55	99.9	23.0	0.02	99.9	32.5	0.02
	init	99.8	2353.4	0.62	99.7	2807.3	0.73	98.2	1923.4	0.50	98.4	2200.5	0.58
	in-use	99.8	1485.6	0.48	99.8	2015.9	0.65	99.6	65.7	0.02	99.6	81.9	0.03
$\mathcal{T}_{\text{policy}}$	*array*	100.0	11.7	0.45	100.0	15.0	0.58	100.0	1.4	0.05	100.0	1.6	0.06
	init	100.0	363.3	0.62	100.0	529.2	0.91	99.3	443.2	0.76	99.0	420.7	0.72
	in-use	100.0	19.4	0.97	100.0	187.9	9.39	70.0	39.0	1.95	70.0	42.5	2.13

Table 5
Proof results and times—policy-specific domain theories

In order to evaluate the effect of redundant axioms in our case, we used a reduced domain theory for the *array*, *init*, and *in-use* safety polices and then re-ran e-setheo-csp03F and Vampire5.0. 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 5 summarizes this experiment and gives the results and times for both the reduced and the original full 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. There is no uniform trend, however—depending on the ATP, the applied simplifications, and the safety policy, either more or less tasks are proven while the proofs can become faster or slower. This non-uniform behavior is likely to be a consequence of the internal architecture of the provers. Both e-setheo and Vampire implement multiple search strategies and then derive a schedule from the proof task. However, e-setheo's scheduling algorithm seems to be more sensitive to the changes than Vampire's. e-setheo never fails to prove proof tasks by switching to the reduced domain theory and sometimes finds a substantial number of additional proofs, in particular for unsimplified or almost unsimplified tasks. The average proof times are usually slightly better but they can vary widely—up to one order of magnitude in both directions (e.g., *init* with $\mathcal{T}_{\text{prop}}$ and *in-use* with $\mathcal{T}_{\text{policy}}$). In contrast, the variation in Vampire's results and proof times is a lot smaller and appears to be statistically insignificant.

5 Proof Checking

For certification purposes, 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 obtain fast results. 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 synthesizer. However, although this idea is very simple in theory, there are currently (as of 2004) almost no proof checkers for high-performance ATPs. This has a number of practical reasons:

- Many ATPs simply do not generate the required detailed proofs, mainly due to implementation effort and run-time requirements.
- On-going changes in the ATP require frequent updates and re-verification of the proof checker.

- 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 the proof checker more complicated and thus hard to verify.
- Almost all ATPs work on problems in CNF, so the proof checking can only be done on that level, and not on the FOL level. Since clausification is often a large part of a proof, this reduces the confidence that proof checking can bring.

The notable exception is the IVY system [19] that we used in our experiments. IVY combines a clausifier and the Otter theorem prover with a proof checker. Because IVY is implemented within the ACL2 logic [16], both the clausifier and the proof checker have been verified. IVY thus provides the same functionality as a verified prover for first-order logic, but achieves relatively good performance by using Otter to find the proofs. However, the formal verification of the IVY clausifier and proof checker are based up finite domains [19] but since the implementation of IVY does not actually rely on the finiteness, the system can be used for arbitrary proof tasks.

Another limitation of IVY is shared by all existing clausification algorithms. Clausifiers usually take a first-order formula apart and reorganize the pieces using non-logical graph-based techniques. Thus, establishing traceability between the clauses (or literals) and the positions they had in the original formula would require substantial effort and has not yet been attempted in practical implementations. While this restriction makes it impossible to translate the clausal proof back into a first-order representation, it also has a negative influence on the prover's behavior. Many ATPs can be sped up considerably if it is known which parts of the formula are axioms and which belong to the conjecture. This distinction allows the prover to apply goal-oriented rules. Our application naturally provides this information, but this is ignored by IVY. Thus, the Otter prover used within IVY can only use Otter's auto-mode which is rather weak for our proof obligations. Experiments also revealed that IVY has problems in handling the full axiom set. With the policy-specific domain theory of Section 4.4, 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 *in-use*.

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 an ATP 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 any places where it is violated.

Safety checks are typically carried out during code reviews [24], 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 code, where each statement is labelled with either "complies with property P", or with information about the violation. This requires two things: (i) tracing information which links the safety obligations (or their proofs) to specific lines of code in the program being certified, and (ii) a summary which relates this detailed information back to the specification and the safety policy, while drawing attention to specific areas of concern.

Existing techniques for addressing the tracing problem [14], however, need to be extended for our purposes. The required information about code locations needs to be threaded through all stages of our certification architecture (cf. Figure 1). Only then can the tracing information be obtained and displayed in the appropriate way. Even if we just want to know if a certain line in the code fulfills a safety property, the location information still needs to be threaded through the VCG and the simplifier.

To get more detailed information, however, the tracing has to be threaded through the ATP and into the proof it generates. For example, the analysis needs to reveal which other lines of code are actually required to satisfy a property. For variable initialization safety this can mean computing on which line the variable that is accessed in the current statement was initialized. The extraction of this information requires knowledge of which parts of the formula contributed to the proof, as well as their location information. This problem is aggravated by the fact that most theorem provers work on clausal normal form, which usually looses the important location information.

In general, useful information extracted from an ATP can be used for purposes of auto-generating documentation. In [6], we describe a *safety documentation* tool, which generates a natural language description explaining the safety of a program, by converting the VCs into text. This could be extended by carrying out some symbolic evaluation from the simplifier as an intermediate step to using the full proofs.

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. 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 5 shows (on the left) the overall results for the different stages and provers.

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 $\mathcal{T}_\emptyset$

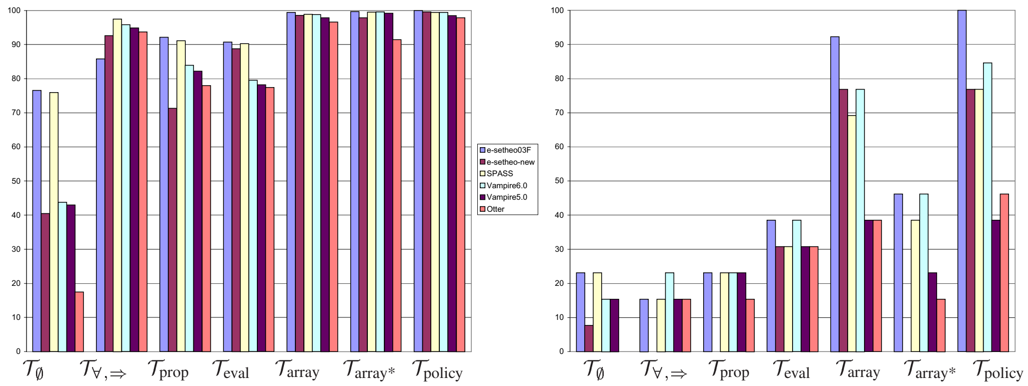

Fig. 5. Comparison of proof results and certification results

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 5 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 larger. 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 hinges on 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 deep 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 length 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 [4] 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 [25]. In another thread of future work we will experiment with other reasoning systems and decision-based tools (such as PVS, Vampire, and Simplify) to process our verification conditions. We also plan to combine our work on certification with automated safety and design document generation [6] tools that we are developing. Finally, we continue to integrate additional safety properties.

References

[1] A. Appel, N. Michael, A. Stump, and R. Virga. "A Trustworthy Proof Checker". *JAR*, 31(3–4):191–229, 2003.

[2] W. Bibel and P. H. Schmitt, (eds.). *Automated Deduction — A Basis for Applications*. Kluwer, 1998.

[3] E. Denney and B. Fischer. "Correctness of Source-Level Safety Policies". In *Proc. FM 2003: Formal Methods*, *LNCS 2805*, pp. 894–913. Springer, 2003.

[4] E. Denney, B. Fischer, and J. Schumann. "Adding Assurance to Automatically Generated Code". In *Proc. 8th IEEE Intl. Sympl. High Assurance System Engineering*, pp. 297–299. IEEE Comp. Soc. Press, 2004.

[5] E. Denney, B. Fischer, and J. Schumann. Using Automated Theorem Provers to Certify Auto-Generated Aerospace Software, 2004. In *Proc. IJCAR'04*. To appear.

[6] E. Denney and R. P. Venkatesan. "A generic software safety document generator". In *Proc. 10th AMAST*. To appear, 2004.

[7] B. Fischer, A. Hajian, K. Knuth, and J. Schumann. Automatic Derivation of Statistical Data Analysis Algorithms: Planetary Nebulae and Beyond. In *Proc. 23rd MaxEnt*, 2004. http://ase.arc.nasa.gov/people/fischer/.

[8] B. Fischer. *Deduction-Based Software Component Retrieval*. PhD thesis, U. Passau, 2001. http://elib.ub.uni-passau.de/opus/volltexte/2002/23/.

[9] C. Flanagan and K. R. M. Leino. "Houdini, an Annotation Assistant for ESC/Java". In *Proc. FME 2001: Formal Methods for Increasing Software Productivity*, *LNCS 2021*, pp. 500–517. Springer, 2001.

[10] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. "Extended static checking for Java". In: *PLDI*, pp. 234–245. ACM, 2002.

[11] B. Fischer and J. Schumann. "Applying AutoBayes to the Analysis of Planetary Nebulae Images". In *Proc. 18th ASE*, pp. 337–342. IEEE Comp. Soc. Press, 2003.

[12] B. Fischer and J. Schumann. "AutoBayes: A System for Generating Data Analysis Programs from Statistical Models". *J. Functional Programming*, 13(3):483–508, 2003.

[13] B. Fischer, J. Schumann, and G. Snelting. "Deduction-Based Software Component Retrieval". Volume III of Bibel and Schmitt [2], pp. 265–292. 1998.

[14] R. Fraer. "Tracing the Origins of Verification Conditions". In *Proc. 5th AMAST*, pp. 241–255, 1996.

[15] P. Homeier and D. Martin. “Trustworthy Tools for Trustworthy Programs: A Verified Verification Condition Generator”. In *Proc. TPHOLS 94*, pp. 269–284. Springer, 1994.

[16] M. Kaufmann and J S. Moore. “An Industrial Strength Theorem Prover for a Logic Based on Common Lisp”. *Software Engineering*, 23(4):203–213, 1997.

[17] D. Loveland. *Automated Theorem Proving: A Logical Basis*. North–Holland, 1978.

[18] J. McCarthy. “Towards a Mathematical Science of Computation”. In *Proc. IFIP Congress 62*, pp. 21–28. North-Holland, 1962.

[19] W. McCune and O. Shumsky. “System description: IVY”. In *Proc. 17th CADE*, *LNAI 1831*, pp. 401–405. Springer, 2000.

[20] W. McCune and L. Wos. “Otter—The CADE-13 Competition Incarnations”. *JAR*, 18(2):211–220, April 1997.

[21] M. Moser, O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann. and K. Mayr. “The Model Elimination Provers SETHEO and E-SETHEO”. *JAR*, 18:237–246, 1997.

[22] G. C. Necula. “Proof-Carrying Code”. In *Proc. 24th POPL*, pp. 106–19. ACM, 1997.

[23] G. C. Necula and P. Lee. “The Design and Implementation of a Certifying Compiler”. In: *PLDI*, pp. 333–344. ACM, 1998.

[24] S. Nelson and J. Schumann. “What makes a Code Review Trustworthy?”. In *Proc. 37th Annual Hawaii International Conference on System Sciences*. IEEE, 2004.

[25] The Programatica Team. “Programatica Tools for Certifiable, Auditable Development of High-assurance Systems in Haskell”. In *Proc. High Confidence Software and Systems Conf.*, Baltimore, MD, April 2003.

[26] W. Reif. “The KIV Approach to Software Verification”. In *KORSO: Methods, Languages and Tools for the Construction of Correct Software*, *LNCS 1009*, pp. 339–370. Springer, 1995.

[27] W. Reif and G. Schellhorn. “Theorem Proving in Large Theories”. Volume III of Bibel and Schmitt [2], pp. 265–292. 1998.

[28] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. *Structured Specifications and Interactive Proofs with KIV*. Volume II of Bibel and Schmitt [2], pp. 225–241, 1998.

[29] A. Riazanov and A. Voronkov. “The Design and Implementation of Vampire”. *AI Communications*, 15(2–3):91–110, 2002.

[30] J. Schumann. *Automated Theorem Proving in Software Engineering*. Springer, 2001.

[31] G. Sutcliffe and C. Suttner. CASC Home Page. `http://www.tptp.org/CASC`.

[32] G. Sutcliffe and C. Suttner. TPTP Home Page. `http://www.tptp.org`.

[33] C. Weidenbach. SPASS Home Page. `http://spass.mpi-sb.mpg.de`.

[34] C. Weidenbach, B. Gaede, and G. Rock. “Spass and Flotter version 0.42”. In *Proc. 13th CADE*, *LNAI 1104*, pp. 141–145. Springer, 1996.

[35] M. Whalen, J. Schumann, and B. Fischer. “AutoBayes/CC — Combining Program Synthesis with Automatic Code Certification (System Description)”. In *Proc. 18th CADE*, *LNAI 2392*, pp. 290–294. Springer, 2002.

[36] M. Whalen, J. Schumann, and B. Fischer. “Synthesizing Certified Code”. In *Proc. FME 2002*, *LNCS 2391*, pp. 431–450. Springer, 2002.

[37] J. Whittle and J. Schumann. Automating the Implementation of Kalman Filter Algorithms, 2004. In review.