Using Automated Theorem Provers to Certify Auto-Generated Aerospace Software

Ewen Denney[†], Bernd Fischer[‡], Johann Schumann[‡]

[†]QSS / [‡]RIACS, NASA Ames Research Center,
{edenney,fisch,schumann}@email.arc.nasa.gov

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). For full automation, however, the obligations must be aggressively preprocessed and simplified. We describe the unique requirements this places on the ATP and demonstrate how the individual simplification stages, which are implemented by rewriting, influence the ability of the ATP to solve the proof tasks. Experiments on more than 25,000 tasks were carried out using Vampire, Spass, and e-setheo.

1 Introduction

Software certification aims to show that the software in question satisfies 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* detailed annotations, e.g., loop invariants. A verification condition generator (VCG) processes the annotated code and produces a set of *safety obligations*, which are provable if and only if the program 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.

In this paper, we describe and evaluate the application of ATPs to discharge the emerging safety obligations. This is a crucial aspect of our approach since its practicability hinges on a very high degree of automation. Our first hypothesis is that the current generation of high-performance ATPs is—in principle—already powerful enough for practical applications. However, this is still a very demanding area because the number of obligations is potentially very large and program verification is generally a hard problem domain for ATPs. Our second hypothesis is thus that the application still needs to carefully preprocess the proof tasks to make them more tractable for ATPs.

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.

We have tested our two hypotheses by running five high-performance provers on seven different versions of the safety obligations resulting from certifying five different safety policies for four different programs—in total more than 25,000 obligations 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 obligations. 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 hypotheses: the provers are generally able to certify all test programs for all polices but only after substantial preprocessing of the obligations. Finally, Section 5 draws some conclusions.

Conceptually, this paper continues the work described in [24, 23] 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 integrated with the AUTOFILTER system [22] as well as with the AUTOBAYES system [8]. We also implemented a new generic VCG which can be customized for an 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 result in a substantially larger experimental basis than reported before.

Related Work KIV [14, 15] is an interactive verification environment which can use ATPs but heavily relies on term rewriting and user guidance. Sunrise [10] is a fully automatic system but uses custom-designed tactics in HOL to discharge the obligations. Flanagan and Leino [6] describe a similar system, Houdini, where the generated proof obligations are discharged by ESC/Java; again, this relies on a significant amount of user interaction.

2 System Architecture

The 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. The synthesis systems take as input a high-level problem specification (cf. Section 3.1 for informal examples). The code that implements the specification is then generated by a schema-based process. *Schemas* are generic algorithms which are instantiated in a problem-specific way after their applicability conditions are proven to hold for the given problem specification. Both systems first generate C++-style intermediate code which is then compiled down into any of the different supported languages and runtime environments. Figure 1 gives an overview of the overall system architecture.

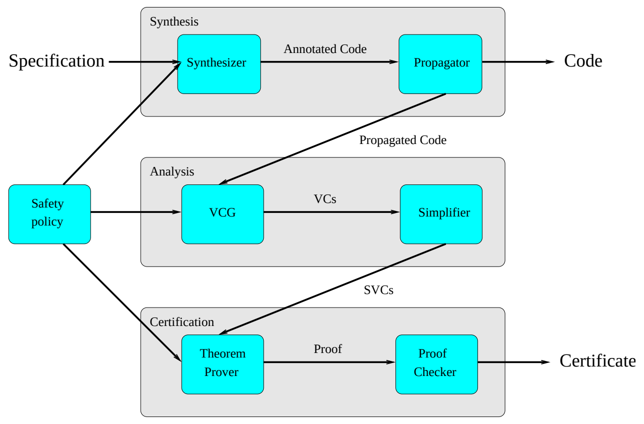

Fig. 1. Certification System Architecture

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 detail in [2].

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*) asserts 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 (*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 c the policy defines a formula $\texttt{cond}(\texttt{c})$, which is the safety condition on

safety policy	safety condition ($\mathtt{cond}(\mathtt{c})$)	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
inuse	$\forall$ *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

Fig. 2. Safety Formulas for Different Policies

c, and a substitution $\mathtt{sub}(\mathtt{c})$, which captures how the command changes the environmental information relevant to the safety policy. Using `cond` and `sub`, the rules of the safety policy can then be derived systematically from the standard Hoare rules of the underlying programming language [2].

From our perspective, the safety conditions $\mathtt{cond}(\mathtt{c})$ are the most interesting aspect since they have the greatest bearing on the form of the proof obligations. Figure 2 summarizes the different formulas 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 requires 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 summation expressions.

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 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 propagated throughout 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, 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 so that it is straightforward to see that it correctly implements the rules of the logic. Hence, the VCG does not carry out any simplifications; in particular, it does not actually apply the substitutions (i.e., execute the specified replacements) but maintains explicit formal substitution terms. Consequently, the generated verification conditions (VCs) tend to be large and must be simplified separately; the more manageable simplified verification conditions (SVCs) which result are then processed by a first order theorem prover. The resulting proofs can be sent to a proof checker (e.g., Ivy [13]). However, since most ATPs do not produce explicit proofs in a standardized format, we will not focus on proof checking here but concentrate on the simplification and theorem proving steps.

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

Fig. 3. Structure of a safety obligation

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. Most of the hypotheses consist of annotations which have been propagated through the code and are irrelevant to the line at hand. The proof obligation also contains the local loop invariants together with bounds on `for`-loops. Finally, the conclusion is generated from the safety formula of the corresponding safety policy.

2.3 Processing Proof Obligations and Connecting the Prover

The simplified safety obligations are then 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 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 was used before invoking the theorem prover. Run-time measurement and prover control (e.g., aborting provers) were performed with the same TPTP tools as in the CASC competition [19].

3 Experimental Setup

3.1 Program Corpus

As 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 1 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 [22]. `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, 4] for more details). `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/ijcar`.

3.2 Simplification

Proof task simplification is an important and integral part of our overall architecture. However, as observed before [9, 5, 17], 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 concentrate on rewrite-based simplifications rather than decision procedures because rewriting is easier to certify: each individual rewrite step can easily be double-checked independently.

Baseline The baseline 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 unfeasible. 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

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

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

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

The choice of the specific orders is arbitrary; choosing for example $<$ instead of $>$ makes no difference. However, a further normalization by elimination of either the partial or the strict order (e.g., using a rule $x \leq y \rightsquigarrow x < y \vee x = y$) leads to a substantial increase in the formula size and thus proves to be counter-productive.

$\mathcal{T}_{\text{eval}}$ 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 (viz. less than five) into Pressburger notation, using rules of the form $x + 1 \rightsquigarrow succ(x)$ and $x - 1 \rightsquigarrow pred(x)$. For many safety policies (e.g., *init*), such terms 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.

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

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 [12], i.e., $\textit{sel}(\textit{upd}(a, i, v), j) \rightsquigarrow i = j \,?\, v : \textit{sel}(a, j)$ for one-dimensional arrays and similar forms for higher-dimensional arrays. Also, 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., $\textit{sel}(\textit{constarray}(v, d), i) \rightsquigarrow v$.

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 $\textit{true} \,?\, x : y \rightsquigarrow x$ and $\textit{false} \,?\, x : y \rightsquigarrow y$ to evaluate conditionals.

In order to evaluate the effect of these domain-specific simplifications properly, 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 situations that frequently arise with specific safety policies. The *init*-policy requires a rule

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

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 $\textit{def} = \textit{use} \rightsquigarrow \textit{false}$, 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

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

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 summations:

$$\begin{array}{l} \sum_{i=0}^{\textit{pred}(0)} x \rightsquigarrow 0 \\ P \wedge x = \sum_{i=0}^{\textit{pred}(n)} Q(i) \Rightarrow x + Q(n) = \sum_{i'=0}^{n} Q(i') \\ \rightsquigarrow P \wedge x = \sum_{i=0}^{\textit{pred}(n)} Q(i) \Rightarrow \sum_{i=0}^{n} Q(i) = \sum_{i=0}^{n} Q(i) \end{array}$$

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. However, both are under explicit control of this specific rewrite rule and not the general rewrite engine, and are implemented directly as Prolog-predicates. A similar rule is required in a very specific situation to substitute an equality into a summation:

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

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., $l \le \textit{rand}(l, u) \le u$.

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 $P\,\mathbf{?}\,T\,\mathbf{:}\,F = R \rightsquigarrow (P \Rightarrow T = R) \wedge (\neg P \Rightarrow F = R)$ and similarly for partial and strict orders. A number of congruence rules move nested occurrences of conditional expressions into the required positions. Finite summations, 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. 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 "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.

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 set of axioms defines the usual properties of equality and the order relations, as well as axioms for simple Pressburger arithmetics and for the domain-specific operators (e.g., $\mathit{sel}/\mathit{upd}$ or *rand*). 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 (in our examples $n \leq 6$), we even generate axioms for explicit successor-terms, i.e., $n = \mathit{succ}^n(0)$ and add a finite induction schema of the form $\forall x\ :\ 0 \leq x \leq n \Rightarrow (x = 0 \vee x = 1 \vee \ldots \vee x = n)$. In our application domain, these axioms are needed for some of the matrix operations; thus n can be limited to the maximal (statically known) size of the matrices.

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 at the CASC-19 [18] proving competition in the FOL-category. We used two versions of e-setheo which have both been derived from the CASC-version. For e-setheo-csp03F, the clausification module has been changed and instead of the clausifier provided by the TPTP toolset [19], FLOTTER V2.1 [21, 20] was used to convert the formulas into a set of clauses. e-setheo-new is a recent development version with several improvements over the original e-setheo-csp03 version. Both versions of Vampire [16] have been taken directly "out of the box"—they are the versions which were running during CASC-19. Spass 2.1 was obtained from the developer's website [20].

In the experiments, we used the default parameter settings and none of the special features of the provers. 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 dual processor standard PC with 4GB memory.

4 Empirical Results

4.1 Generating and Simplifying Obligations

Table 1 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 respective size of the generated annotations (before propagation), and then, for each simplifier, the elapsed time and the number of generated obligations.

Example	LoC	Policy	LoA	$\mathcal{T}_\emptyset$		$\mathcal{T}_{\forall,\Rightarrow}$		$\mathcal{T}_{\text{prop}}$		$\mathcal{T}_{\text{eval}}$		$\mathcal{T}_{\text{array}}$		$\mathcal{T}_{\text{array}^*}$		$\mathcal{T}_{\text{policy}}$	
`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
		inuse	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
		inuse	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
`segm`	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
`gauss`	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 1. Results of generating safety 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 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 respective 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 arithmetics ($\mathcal{T}_{\text{eval}}$) 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` even 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 only, $\mathcal{T}_{\text{array*}}$, 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 less 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 2 summarizes the results obtained from running the theorem provers on all proof obligations (except for the invalid obligations from the *inuse*-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*, *inuse*, *symm*, and *norm*). Then, for each prover, the percentage of solved proof obligations and the total CPU-time are given. The last two columns give the maximum and minimum percentage of solved tasks.

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-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 $\mathcal{T}_{\text{array}}$ 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 $\mathcal{T}_{\text{array*}}$, which only contains the language-specific rules, also illustrates this behavior. The *norm*-policy clearly produces the most difficult proof 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 2 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 [11]. Flotter has a highly elaborate conversion algorithm which performs many simplifications and

			e-setheo03F		e-setheo-new		SPASS		Vampire6.0		Vampire5.0		max/min	
Simp	Pol	N	%	T_{proof}	%	T_{proof}	%	T_{proof}	%	T_{proof}	%	T_{proof}	%	%
$\mathcal{T}_{\emptyset}$	array	110	96.4	192.4	94.5	284.9	96.4	73.4	95.5	178.1	95.5	102.1	96.4	94.5
	init	164	76.8	3000.8	13.1	1759.8	75.0	2898.3	8.5	9224.9	8.5	8251.0	76.8	8.5
	in-use	19	57.9	610.8	44.4	612.2	68.4	512.8	57.9	773.1	47.4	645.5	68.4	44.4
	symm	18	50.0	387.7	8.3	266.1	38.9	555.3	16.7	744.9	16.7	723.6	50.0	8.3
	norm	54	51.9	1282.4	51.9	1341.0	51.9	1224.2	50.0	1316.5	48.1	1327.1	51.9	48.1
$\mathcal{T}_{\forall,\Rightarrow}$	array	1457	99.0	903.4	94.2	5925.0	99.8	217.0	99.9	240.5	99.8	152.4	99.9	94.2
	init	3177	88.4	3969.4	91.7	20784.8	97.4	8732.2	95.0	14482.2	93.5	14203.4	97.4	88.4
	in-use	1123	59.3	819.1	96.4	4100.3	99.1	1733.5	95.3	4183.7	94.3	4206.8	99.1	59.3
	symm	286	93.4	1785.9	90.6	2341.0	88.5	3638.7	90.2	3315.8	91.3	1789.2	93.4	88.5
	norm	155	85.8	1422.1	73.5	2552.5	84.5	1572.0	87.7	1359.9	87.1	1276.0	87.7	73.5
$\mathcal{T}_{\text{prop}}$	array	275	99.3	278.2	76.4	4080.8	99.3	157.5	99.3	187.5	99.3	132.6	99.3	76.4
	init	919	94.7	4239.4	73.0	17472.2	92.8	5469.7	84.9	10598.0	83.2	10546.8	94.7	73.0
	in-use	177	86.4	1854.0	77.4	2768.2	94.9	1008.3	70.1	3806.2	65.0	3960.6	94.9	65.0
	symm	56	66.1	1476.2	51.8	1944.4	48.2	1911.3	58.9	1596.7	58.9	1424.8	66.1	48.2
	norm	41	46.3	1361.2	0.0	2483.3	41.5	1478.2	53.7	1286.7	51.2	1275.3	53.7	0.0
$\mathcal{T}_{\text{eval}}$	array	28	100.0	16.2	100.0	19.7	100.0	10.4	100.0	12.7	100.0	1.7	100.0	100.0
	init	790	94.6	3944.2	94.1	8288.0	93.3	4380.1	82.5	10239.0	82.0	9040.2	94.6	82.0
	in-use	172	86.0	1852.2	83.1	2305.2	94.8	1023.1	69.8	3718.1	67.4	3561.1	94.8	67.4
	symm	56	66.1	1451.1	66.1	1500.4	51.8	1716.0	62.5	1455.5	58.9	1389.8	66.1	51.8
	norm	30	53.3	859.4	13.3	1575.8	50.0	940.5	66.7	736.7	53.3	858.0	66.7	13.3
$\mathcal{T}_{\text{array}}$	array	28	100.0	15.4	100.0	19.8	100.0	10.4	100.0	12.7	100.0	1.7	100.0	100.0
	init	582	100.0	527.6	100.0	823.9	99.7	875.8	100.0	1401.3	99.0	785.1	100.0	99.0
	in-use	47	100.0	323.9	100.0	343.2	100.0	171.3	100.0	262.6	87.2	525.2	100.0	87.2
	symm	1337	100.0	1104.3	99.9	1629.3	99.4	746.4	99.1	963.9	99.0	922.7	100.0	99.0
	norm	32	59.4	678.4	18.8	1583.1	59.4	709.7	62.5	791.7	50.0	858.6	62.5	18.8
$\mathcal{T}_{\text{array*}}$	array	1457	99.9	916.4	94.2	5918.0	99.9	210.8	99.9	240.6	99.9	153.1	99.9	94.2
	init	3825	99.7	3412.3	96.3	13536.1	99.5	4574.9	99.8	4952.1	98.4	6000.1	99.8	96.3
	in-use	3089	99.8	2438.4	99.4	5139.0	99.8	889.2	99.8	793.5	99.6	925.9	99.8	99.4
	symm	6403	99.9	5317.4	99.7	11787.7	99.7	3385.1	99.6	3277.3	99.6	1807.0	99.9	99.6
	norm	157	86.0	1306.8	72.6	2670.8	86.0	1351.3	86.6	1449.9	86.0	1276.2	86.6	72.6
$\mathcal{T}_{\text{policy}}$	array	26	100.0	15.0	100.0	17.7	100.0	9.9	100.0	12.0	100.0	1.6	100.0	100.0
	init	582	100.0	529.2	100.0	827.9	99.5	875.2	100.0	1418.9	99.0	782.5	100.0	99.0
	in-use	20	100.0	281.7	100.0	329.7	100.0	170.7	100.0	262.6	70.0	524.8	100.0	70.0
	symm	1345	100.0	1104.6	99.9	1640.5	99.4	760.0	99.1	1048.8	99.0	926.9	100.0	99.0
	norm	14	100.0	9.0	57.1	375.8	100.0	26.2	100.0	108.0	71.4	241.8	100.0	57.1

Table 2. Certification results and times

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 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 "hand-picked" for certain properties, a large number of them is (almost) trivial—even in the unsimplified case. Figure 4 shows the resources required for the proof tasks as a series of pie charts. For each simplification stage, we show the percentage of proof tasks which are "very simple" (i.e., $T_{proof} < 1s$, white), those which require between 1 and 10 seconds of run time (light gray), and the difficult ones (dark gray). The percentage of tasks which fail within the 60s time limit is displayed in black. All numbers are obtained with e-setheo-csp03F; the figures for the other provers look similar. With additional preprocessing and simplification of the proof obligations, the number of failing proof tasks decreases sharply from approximately 16% to zero and the number of easy tasks increases substantially. This demonstrates the advantages of aggressive simplification of the proof tasks.

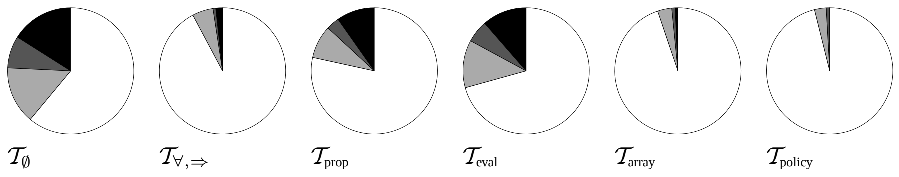

Fig. 4. Distribution of short ($T_{proof} < 1s$, white), medium ($T_{proof} < 10s$, light grey), long ($T_{proof} < 60s$, dark grey) proofs, and failing tasks (black) for the various simplification stages (e-setheo-csp03F).

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 similarities in structure. We have identified three classes of hard examples; these classes are directly addressed by the rewrite rules of the policy-specific simplifications (see Section 3.2).

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 (as defined in Section 2.1) 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:

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

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

$$\forall i, j \cdot 0 \leq i < n \wedge 0 \leq j \leq n \Rightarrow (i \neq 0 \wedge j \neq 0 \wedge \ldots i \neq n \wedge j \neq n \Rightarrow \mathit{false})$$

and soon become intractable for the clausifier, even for small n ($n = 2$ or $n = 3$).

5 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 obligations generated by our system. With "out-of-the-box" provers, only about two-thirds of the obligations could be proven. However, after aggressive simplification, most of the provers could solve all emerging obligations. In order to see the effects of simplification more clearly, we experimented with specific preprocessing stages.

It is well-known that, in contrast to traditional mathematics, software verification hinges on large numbers of mathematically shallow 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.

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. Although specific parameter settings in a prover might overcome this problem, this would require a deep knowledge of the individual theorem provers, and in our experiments, we did not use any specific features or parameter settings for the individual theorem provers.

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 [3] for a comparison).

Our current efforts focus on integrating additional safety properties and extending the approach to synthesized code that has been modified manually.

References

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

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

[3] E. Denney, B. Fischer, and J. Schumann. “Adding Assurance to Automatically Generated Code”. Accepted for High Assurance System Engineering 2004.

[4] B. Fischer, A. Hajian, K. Knuth, and J. Schumann. Automatic Derivation of Statistical Data Analysis Algorithms: Planetary Nebulae and Beyond, 2003. Accepted for publication. `http://ase.arc.nasa.gov/people/fischer/`.

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

[6] 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.

[7] 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.

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

[9] B. Fischer, J. Schumann, and G. Snelting. “Deduction-Based Software Component Retrieval”. In Bibel and Schmitt [1], pp. 265–292.

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

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

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

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

[14] 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.

[15] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. *Structured Specifications and Interactive Proofs with KIV*, chapter II.1, pp. 13–40. Volume II of Bibel and Schmitt [1], 1998.

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

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

[18] G. Sutcliffe and C. Suttner. CASC 19, 2003.
`http://www.cs.miami.edu/~tptp/CASC/19`.

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

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

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

[22] J. Whittle and J. Schumann. Automating the Implementation of Kalman-Filter Algorithms, 2003. In review.

[23] 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.

[24] M. Whalen, J. Schumann, and B. Fischer. “Synthesizing Certified Code”. In *Proc. FME 2002: Formal Methods—Getting IT Right*, *LNCS 2391*, pp. 431–450. Springer, 2002.