Certifiable Program Generation

Ewen Denney and Bernd Fischer

USRA/RIACS, NASA Ames Research Center, Moffett Field, CA 94035, USA
{edenney, fisch}@email.arc.nasa.gov

Abstract. Code generators based on template expansion techniques are easier to build than purely deductive systems but do not guarantee the same level of assurance: instead of providing "correctness-by-construction", the correctness of the generated code depends on the correctness of the generator itself. We present an alternative assurance approach, in which the generator is extended to enable Hoare-style safety proofs for each individual generated program. The proofs ensure that the generated code does not "go wrong", i.e., does not violate certain conditions during its execution.

The crucial step in this approach is to extend the generator in such way that it produces all required annotations (i.e., pre-/postconditions and loop invariants) without compromising the assurance provided by the subsequent verification phase. This is achieved by embedding annotation templates into the code templates, which are then instantiated in parallel by the generator. This is feasible because the structure of the generated code and the possible safety properties are known when the generator is developed. It does not compromise the provided assurance because the annotations only serve as auxiliary lemmas and errors in the annotation templates ultimately lead to unprovable safety obligations.

We have implemented this approach and integrated it into the AUTOBAYES and AUTOFILTER program generators. We have then used it to fully automatically prove that code generated by the two systems satisfies both language-specific properties such as array-bounds safety or proper variable initialization-before-use and domain-specific properties such as vector normalization, matrix symmetry, or correct sensor input usage.

1 Introduction

Program generation has a significant potential to improve the software development process and promises many benefits, including higher productivity, reduced turn-around times, increased portability, and elimination of manual coding errors. However, the key to realizing these benefits is of course generator correctness—nothing is gained from replacing manual coding errors with automatic coding errors. Moreover, following the motto "trust but verify" there should be some more explicit evidence for the correctness of the generated code than just trust in the correctness of the generator itself, or as has been argued, "rigorous arguments must be provided to demonstrate the correctness of the translator and/or the generated code" [WH99].

Several approaches have been explored to ensure and demonstrate correctness. In deductive program synthesis [SW$^+$94, Kre98], the program is generated as byproduct of an existence proof for a theorem derived from the specification; other approaches based on refinement [Smi90, BG$^+$98] or translation verification [WB96] can offer similar "correct-by-construction" guarantees. However, code generators based on these ideas are difficult to build and to scale up, and have not found widespread application. Code generators that are based on template expansion techniques are easier to build but can currently not guarantee the same level of assurance. Traditionally, they are only validated by testing, which requires significant effort that can quickly become excessive, in particular for applications in high-assurance domains like aerospace. For example, the aerospace software development standard DO-178B [RTC92] mandates that the implementation of the generator is tested to the same level of criticality the generated code requires.

We are developing and implementing an alternative approach that is not based on the verification or validation of the generator but instead focuses on the safety of each individual generated program. Our core idea is to extend the generator itself such that it produces all logical annotations (i.e., pre-/postconditions and loop invariants) that are required for formal safety proofs in a Hoare-style framework. These proofs certify that the generated code does not "go wrong", i.e., does not violate certain conditions during its execution. The crucial aspect of the approach is to ensure that errors in the original code generator or in the certification extension do not compromise the assurance provided by the subsequent verification phase, or, in other words, that the proofs are correct and actually prove the safety properties claimed.

We have integrated this approach into the program generators AUTOBAYES [FS03] and AUTOFILTER [WS04]. We have then used it to fully automatically prove that code generated by the two systems satisfies both language-specific properties such as array-bounds safety or proper variable initialization-before-use and domain-specific properties such as vector normalization, matrix symmetry, or correct sensor input usage.

This paper summarizes our previous work on certifiable program generation. More details can be found in the cited references.

2 AUTOBAYES and AUTOFILTER

AUTOBAYES and AUTOFILTER are two domain-specific program generators that follow a schema-based approach to code generation. This extends the "plain" template expansion techniques by adding semantic constraints to the templates. AUTOBAYES [FS03] works in the scientific data analysis domain and generates parameter learning programs, while AUTOFILTER [WS04] generates state estimation code based on variants of the Kalman filter algorithm. Both systems share a large common core (e.g., symbolic subsystem, certification subsystem, and target code generators) but have their individual schema libraries. They are implemented in SWI-Prolog and together comprise approximately 100 kLoC. Both systems work fully automatically and can generate code of considerable size and complexity (approximately 1500 LoC with deeply nested loops) within a few seconds.

Schemas. A *schema* comprises a parameterized code fragment (i.e., template) together with a set of constraints that determine whether the schema is applicable and how the parameters can be instantiated. The constraints are formulated as conditions on a problem model, which allows the problem structure to directly guide the application of the schemas and thus constrains the search space. The parameters are instantiated by the code generator, either directly on schema application or by recursive calls with a modified problem. The schemas are organized hierarchically into a *schema library* which further constrains the search space. Schemas represent both fundamental building blocks (i.e., algorithms) and solution methods (i.e., transformations) of the domain; they are thus similar to the lemmas used in purely deductive systems but they can contain explicit calls to a meta-programming kernel in order to construct code.

Symbolic Computations. Symbolic computations are used in AutoBayes and AutoFilter to support schema instantiation and code optimization. The core of the symbolic subsystem is a small rewrite engine which supports associative-commutative operators and explicit contexts. It thus allows rules as for example $x/x \rightarrow_{C \vdash x \neq 0} 1$ where $\rightarrow_{C \vdash x \neq 0}$ means "rewrites to, provided $x \neq 0$ can be proven from the current context C." Expression simplification and symbolic differentiation are implemented on top of the rewrite engine. The basic rules are straightforward; however, vectors and matrices require careful formalizations, and some rules also require explicit meta-programming, e.g., when bound variables are involved.

Intermediate Code. The code fragments in the schemas are formulated in an imperative intermediate language. This is essentially a "sanitized" variant of C (i.e., no pointers, no side effects in expressions etc.); however, it also contains a number of domain-specific constructs like vector/matrix operations, finite sums, and convergence-loops.

Optimization. Straightforward schema instantiation and composition produces suboptimal code; worse, many of the suboptimalities cannot be removed completely using a separate, after-the-fact optimization phase. Schemas can thus explicitly trigger large-scale optimizations which take into account information from the code generation process. For example, all numeric routines restructure the goal expression using code motion, common sub-expression elimination, and memoization; since the schemas know the goal variables, no dataflow analysis is required to identify invariant sub-expressions, and code can be moved around aggressively, even across procedure borders.

Target Code Generation. In a final step, the optimized intermediate code is translated into code tailored for a specific run-time environment. We currently have target code generators for the Octave and Matlab environments, and can also produce standalone Ada, C, and Modula-2 code. Each target code generator employs one rewrite system to eliminate the constructs of the intermediate language which are not supported by the target environment ("desugaring") and a second rewrite system to clean up the desugared code; most rules are shared between the different code generators.

Problem Specifications. Schema-based program generation does not necessarily require a logical conjecture as starting point for a proof. The Code derivation can therefore begin with a specification in a more application-oriented *domain-specific language*. Our specification languages combine some target language constructs (e.g., declarations) with established scientific and engineering notations (e.g., differential equations). This allows a concise and fully declarative formulation of the problem together with some details of the desired configuration and architecture of the code to be generated. AUTOBAYES uses a specification language that is very close to the generative statistical models used in Bayesian statistics, while AUTOFILTER uses a more control engineering-oriented notation to formulate process models.

3 Certification Architecture

Our certification approach generally follows similar lines as proof carrying code (PCC) [Nec97]; in particular, the role of the extended code generator as producer of annotated target code is very similar to that of a certifying compiler [NL98, CL^{+}00] in the PCC approach. However, there are also some key differences. First, since we target code generation instead of compilation, we work on the source code level instead of the object code level. On the positive side, since some safety properties can be formulated more naturally (e.g., initialization-before-use) or only (e.g., loop variable restrictions) on the source code level, this allows us to formulate and support more safety properties relevant to application domains. In particular, high-level domain-specific properties such as matrix symmetry or frame safety [LPR01] are inherently defined on the source code level. On the negative side, these domain-specific properties make the annotation generation (see Section 5) more difficult. Fortunately, unlike a general purpose compiler, a domain-specific code generator embodies enough domain knowledge to provide the information required. Second, the proofs are not tightly integrated into the code and are currently not even distributed together with the code; hence, our approach provides *certifiable* rather than *certifying* program generation. However, this is not a fundamental deficit and could be changed relatively straightforwardly, if necessary. Third, we apply a different prover technology and our architecture allows choosing from different off-the-shelf fully automated theorem provers (ATP) for first-order logic instead of relying on a customized higher-order system. However, the ATP can essentially be considered as a black box.

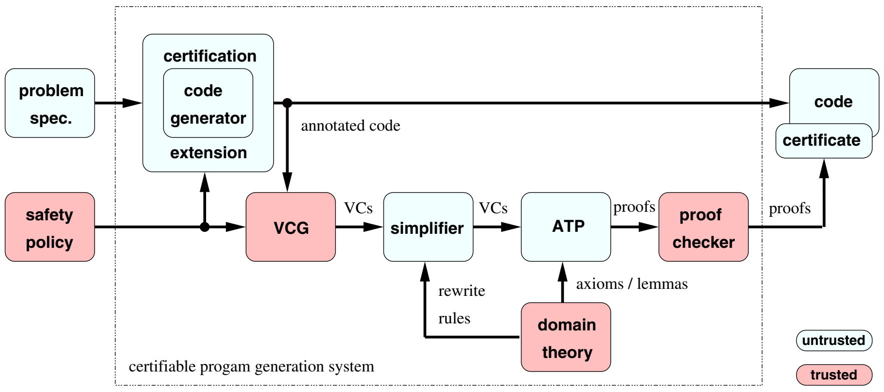

Fig. 1. Certifiable program generation: System architecture

Figure 1 shows the overall architecture of a certifiable program generation system. At its core is the original code generator which is extended for certification purposes and complemented by a verification condition generator (VCG), a simplifier, an ATP, a proof checker, and a domain theory. These components and their interactions are described in the rest of this paper and in more detail in [WSF02, DF03, DFS05]. As in the PCC approach, the architecture distinguishes between trusted and untrusted components, shown in Figure 1 in red (dark grey) and blue (light grey), respectively. *Trusted* components *must be correct* because any errors in them can compromise the assurance provided by the overall system. *Untrusted* components, on the other hand, are not crucial to the assurance because their results are double-checked by at least one trusted component. In particular, the assurance provided by a certifiable program generation system does not depend on the correctness of its two largest components: the original code generator (including the certification extensions), and the ATP; instead, we need only trust the safety policy, the VCG, the domain theory, and the proof checker.

4 Source-Level Safety Certification

The purpose of safety certification is to demonstrate that the code does not violate certain conditions during its execution. A *safety property* is an exact characterization of these conditions based on the operational semantics of the language. It is formally defined as an entailment relation that formalizes when the evaluation of an expression and the execution of a statement are safe in a given environment. A *safety policy* is a set of proof rules and auxiliary definitions which are designed to show that safe programs satisfy the safety property of interest. The intention is that a safety policy enforces a particular safety property and strictly speaking an off-line proof is required to show the policy correct with respect to the property [DF03]. Since the calculus is sound and complete (modulo the completeness of the logic underlying the formulation of the annotations), it does in particular prevent us from proving unsafe programs safe. The proof rules can be formalized concisely using the usual Hoare triples $P\ \{c\}\ Q$, i.e., if the condition P holds before and the command c terminates, then Q holds afterwards (see [Mit96] for more information about Hoare-style program proofs).

For each notion of safety which is of interest a safety property and the corresponding safety policy need be formulated. The formulation of the safety property is usually straightforward, and the proof rules for any given safety policy can fortunately be constructed systematically, by instantiating a generic rule set that is derived from the standard rules of the Hoare-calculus [DF03]. The basic idea is to extend the standard environment of program variables with a “safety environment” of “safety” or “shadow” variables which record safety information related to the corresponding program variable. The rules are then responsible for maintaining this environment and producing the appropriate safety obligations.

Figure 2 shows the rules instantiated for the relatively simple case of memory safety. Here the safety environment consists of shadow variables x_{hi} that are used to record the dimension of the corresponding arrays x. The only statement that affects the value of a shadow variable is thus the declaration of an array (cf. the *adecl*-rule). However, all rules also need to produce the appropriate safety formulas $\mathit{safe}_{\text{mem}}(e)$ for all immediate subexpressions e of the statements. Since the safety property defines that an expression is safe if all access to array variables are within the bounds given by the corresponding shadow variables, $\mathit{safe}_{\text{mem}}(x\,[\,e\,]\,)$ for example simply translates to $1 \leq e \leq x_{\text{hi}}$.

$$(\mathit{decl})\quad \frac{}{Q\ \{\textbf{var}\ x\}\ Q}$$

$$(\mathit{adecl})\quad \frac{}{Q[n/x_{\text{hi}}]\ \{\textbf{var}\ x\,[\,n\,]\,\}\ Q}$$

$$(\mathit{skip})\quad \frac{}{Q\ \{\textbf{skip}\}\ Q}$$

$$(\mathit{assign})\quad \frac{}{Q[e/x] \wedge \mathit{safe}_{\text{mem}}(e)\ \{x\ :=e\}\ Q}$$

$$(\mathit{update})\quad \frac{}{Q[\mathit{upd}(x,e_1,e_2)/x] \wedge \mathit{safe}_{\text{mem}}(x\,[\,e_1\,]\,) \wedge \mathit{safe}_{\text{mem}}(e_2)\ \{x\,[\,e_1\,]\ :=e_2\}\ Q}$$

$$(\mathit{if})\quad \frac{P_1\ \{c_1\}\ Q \quad P_2\ \{c_2\}\ Q}{(b \Rightarrow P_1) \wedge (\neg b \Rightarrow P_2) \wedge \mathit{safe}_{\text{mem}}(b)\ \{\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\}\ Q}$$

$$(\mathit{while})\quad \frac{P\ \{c\}\ I \quad I \wedge b \Rightarrow P \quad I \wedge \neg b \Rightarrow Q}{I \wedge \mathit{safe}_{\text{mem}}(b)\ \{\textbf{while}\ b\ \textbf{inv}\ I\ \textbf{do}\ c\}\ Q}$$

$$(\mathit{for})\quad \frac{P\ \{c\}\ I[i+1/i] \quad I \wedge e_1 \leq i \leq e_2 \Rightarrow P \quad I[e_2+1/i] \Rightarrow Q}{I[e_1/i] \wedge \mathit{safe}_{\text{mem}}(e_1) \wedge \mathit{safe}_{\text{mem}}(e_2)\ \{\textbf{for}\ i\ :=e_1\ \textbf{to}\ e_2\ \textbf{inv}\ I\ \textbf{do}\ c\}\ Q}$$

$$(\mathit{comp})\quad \frac{P\ \{c_1\}\ R \quad R\ \{c_2\}\ Q}{P\ \{c_1\ ;\ c_2\}\ Q}$$

$$(\mathit{assert})\quad \frac{P \Rightarrow P' \quad P\ \{c\}\ Q' \quad Q' \Rightarrow Q}{P\ \{\textbf{pre}\ P'\ c\ \textbf{post}\ Q'\}\ Q}$$

$$(\mathit{cons})\quad \frac{P \Rightarrow P' \quad P'\ \{c\}\ Q' \quad Q' \Rightarrow Q}{P\ \{c\}\ Q}$$

Fig. 2. Proof rules for memory safety

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

The VCG directly implements the generic rules and, starting with the initial postcondition *true*, applies them to the statements in the usual backwards style, emitting the constructed safety obligations along the way. Since it is part of the trusted component base, it has been designed to be "correct-by-inspection", i.e., deliberately simple. Hence, it does not implement any optimizations or even apply any simplifications. Consequently, the generated obligations tend to be large and must be simplified separately before they can be tackled by the ATP. The resulting proofs can be sent to a proof checker to ensure that the ATP produced a valid proof.

5 Annotation Generation

As Figure 2 shows, the proof rules require logical annotations, in particular loop invariants. The construction of these annotations is usually the limiting factor for the practical application of Hoare-style verification tools, e.g., ESC [FL$^+$02]. Fortunately, the annotations are untrusted. They are never directly used as safety obligations themselves but only serve as lemmas for use by the trusted VCG. Due to the soundness of the calculus, an error in an annotation can, at worst, lead to the construction of an invalid safety obligation. As an example, consider the *while*-rule. If the constructed invariant I is too strong (e.g., $I \equiv$ *false*), then it is easy to show that the postcondition Q follows but impossible to show that the precondition P constructed from the loop body c holds. If the invariant is too weak (e.g., $I \equiv$ *true*), then it will generally be impossible to show that either the precondition or the postcondition follows, unless of course the program is trivially safe. Similar arguments hold for the other rules as well. Even compatible "parallel" errors in the generation of the code and the annotations will not compromise the assurance. In the worst case, the generated code will be functionally wrong but if the proof succeeds, it will still execute safely. This role of the annotations thus allows us to extend the untrusted code generator to produce both code *and* annotations, without compromising the assurance provided by the safety proofs.

The central question though is how this can be achieved. Obviously, there is no free lunch, and ultimately the annotations have to be provided by the developer. This can be done in the form of annotation templates that are integrated into the schemas and instantiated in parallel with the code templates by the generator. The basic process to extend the generator comprises the following four steps:

1. Analyze the generated code and identify the location and structure of the required annotations.
2. For each location, identify the schemas that produced the respective code fragment.
3. For each affected schema, generalize the respective annotations to appropriate meta-annotations (e.g., replace program variables by meta variables).
4. For each meta-annotation, formulate an appropriate annotation template or meta-program that generates the annotation at the time of schema application, and integrate it into the schema.

Like building the generator in the first place, this an expensive manual process. It has to be repeated until all schemas are covered, and started again for each new safety property. Moreover, the annotations are cross-cutting concerns, not only on the level of the generated programs, but also on the level of the program generator. This can make the extension of the generator quite hard work. However, it remains feasible because the overall structure and the purpose of the generated code as well as the possible safety properties are already known when the generator is extended.

```
          ...
          var $a[n]$ ; var $b[m]$
          ...
/* A: */  for $i$ := 1 to $n$
          inv $\forall j \cdot 1 \leq j < i \Rightarrow 1 \leq a[j] \leq m$
           do $a[i]$ := $f(i)$ ;
          post $\forall i \cdot 1 \leq i \leq n \Rightarrow 1 \leq a[i] \leq m$
          ...
/* B: */  for $i$ := 1 to $n$
          inv $1 \leq a[i] \leq m$
           do $b[a[i]]$ := $g(i)$ ;
          ...
```

Fig. 3. Code fragment with annotations

The code fragment shown in Figure 3, which is taken in simplified form from code generated by AUTOBAYES, illustrates how the process works. It uses the loop at A to initialize the array a with an unspecified expression $f(i)$, and then uses a in the second loop at B to write the also unspecified expression $g(i)$ indirectly into b. In order to prove these array accesses safe, the invariant needs to restrict the contents of the a-elements to the valid index range of b, i.e., $a[i]$ has to be between 1 and m.

The question is now how and where to construct this invariant. However, when we write the schema that generates the first loop, we also already know that the a-elements will be used to index into b. This is part of the domain knowledge that is required to build the original code generator. In a first step, we thus extend this schema by an annotation template or meta-program that constructs the (local) invariant and postcondition given at A. The annotation template can focus on the locally relevant information, without needing to describe all the global information that may later be necessary for the proofs because the schemas are not combined arbitrarily but only along the hierarchy given in the schema library.

Unfortunately, these local annotations are in general still insufficient to prove the postcondition at the end of larger code fragments. In our example, we still need to get the information about the values in a into the loop invariant at B. Since this limited information transport is a recurrent problem, we do not pass around the constructed annotations during generation, but rely on a separate *annotation propagation* phase after the code has been constructed. The propagation algorithm can be seen as a very crude approximation of a strongest postcondition predicate transformer. It pushes the generated local annotations forward along the edges of the syntax tree as long as the information can be guaranteed to remain unchanged. Because the generator produces code with restricted aliasing only, the test for which statements influence which annotations can easily be accomplished without a full static analysis by maintaining a set of modified variables during propagation.

The propagation phase also adds a few default annotations as it traverses the code, for example bounds on the loop variables. These could in principle also be reconstructed by the VCG, but that would complicated the implementation of the trusted VCG. The fully annotated and propagated code is then used by the VCG.

6 Experimental Results

We have used the approach described here to certify different safety properties for code generated by AUTOBAYES and AUTOFILTER. Table 1 summarizes the relevant numbers for four representative examples. The first two examples are AUTOFILTER specifications. `ds1` is taken from the attitude control system of NASA's Deep Space One mission [WS04]. `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 [FH$^+$03] of planetary nebula images taken by the Hubble Space Telescope. `segm` describes an image segmentation problem for which an iterative numerical 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, it contains many deeply nested loops, and some of them do not have a fixed (i.e., known at generation time) number of iterations but are executed until a dynamically calculated error value becomes small enough. In contrast, in the Kalman filter code, all loops are executed a fixed number of times. Second, all array accesses are element by element and there are no operations on entire matrices (e.g., matrix multiplication).

For each of the examples, Table 1 lists the size $|S|$ of the specification, the size $|P|$ of the generated program (including comments but without annotations), the applicable safety policies, the sizes $|A|$ and $|A^*|$ of the generated and propagated annotations, and finally the numbers N and N_{fail} of generated and invalid safety obligations as well as the generation and proof times T_{gen} and T_{proof}. All times are wall-clock times rounded to the next second and were obtained on a 2.4GHz standard Linux PC with 4GB memory. The generation times also include generation, simplification, and file output of the safety obligations; code generation alone accounts for approximately 90% of the times listed under the *array* safety policy. The proof times are based on using the E-Setheo [MI$^+$97] prover which was able to discharge all valid obligations; they do not include the time spent on the invalid obligations. A more detailed analysis of the results achieved with different ATPs is available in [DFS05].

The table shows that the generated annotations can amount to a significant fraction of the generated code and, after propagation, can even dominate it. It also shows substantial differences in the size of the annotations required for the different safety properties; in particular, it also shows that array-bounds safety (which is the core prop-

Table 1. Results of safety certification

Example	$\|S\|$	$\|P\|$	Policy	$\|A\|$	$\|A^*\|$	N	N_{fail}	T_{gen}	T_{proof}
`ds1`	48	431	*array*	0	19	1	-	6	1
			init	87	444	74	-	11	84
			inuse	61	413	21	1	8	202
			symm	75	261	865	-	71	794
`iss`	97	755	*array*	0	19	4	-	25	3
			init	88	458	71	-	40	88
			inuse	60	361	1	1	32	-
			symm	87	274	480	-	66	510
`segm`	17	517	*array*	0	53	1	-	3	1
			init	171	1090	121	-	8	109
			norm	195	247	14	-	4	12
`gauss`	18	1039	*array*	20	505	20	-	21	16
			init	118	1615	316	-	54	259

erty guaranteed by PCC) requires almost no local annotations and can often be certified with only the default annotations added by the propagator. The number of generated safety obligations also varies substantially for the different safety properties. However, the proof effort remains tractable, and in most of the cases the ATP was able to successfully discharge all obligations in less than 15 minutes wall clock time.

In general, of course, an obligation can fail to be proven for a number of reasons. First, there may of course be an actual safety violation in the code. This is the case for the two invalid obligations that are produced for the sensor input usage property. The deeper reason for this, however, is not a flaw in the code generator but a sloppy specification that declares a vector that is not completely used. Second, the (generated) annotations may be insufficient or wrong. Annotation errors can come from any part of the schema, or from the propagation phase: an annotation might not be propagated far enough, or it might be propagated out of scope. Third, the theorem prover may time-out, either due to the size and complexity of the obligation, or due to an incomplete domain theory. For certification purposes, however, it is important to distinguish between unsafe programs and any other reasons for failure, and in the case of genuine safety violations, to locate the unsafe parts of the program.

7 Conclusions

We have described an extension to the AUTOBAYES and AUTOFILTER program generators which can automatically ensure important safety properties for the generated code. The core idea of our approach is to extend the generator itself in such way that it produces all logical annotations (i.e., pre-/postconditions and loop invariants) required for Hoare-style safety proofs without compromising the assurance provided by the proofs. In principle, the prover can fail to prove some valid proof obligations and thus raise false alarms but in practice we were able to design the system such that all valid obligations could be discharged fully automatically.

Our approach can be seen as "PCC for code generators" because it enables safety proofs for the generated code. We believe that it can directly be applied to code generators based on template expansion techniques in general, not only to our own systems. However, we believe also that our techniques could as well be used in a response to the recently announced Grand Challenge of developing a verifying compiler. We further believe that in principle any verification technique that can be guided by an appropriate form of annotations can be combined successfully with a certifiable code generator, not just the Hoare-style certification using the VCG/ATP combination described here. Another interesting research direction would thus be to combine annotation generation with other techniques, for example static analysis.

For future work, we plan to extend the system in two main areas, in addition to continually increasing the systems' generative power with more algorithmic schemas, more specification features, and more control over the derivation.

First, we are developing a more declarative and explicit modeling style. Much of the domain knowledge used by the system in deriving code is currently implicit; by making it explicit this can be used to (among other things) facilitate traceability between the code and it derivation in the generated documentation.

Second, we continue to extend the certification power of the system with more policies, more automation, and an integrated approach to documentation generation. In particular, we are now developing an "annotation inference" technique which addresses many of the difficulties of annotation generation. This will also enable us to easily apply our certification techniques to code generators other than our own.

Acknowledgements. Mike Whalen and Johann Schumann contributed substantially to the development and implementation of the certifiable program generation approach.

References

[BG+98] L. Blaine, L.-M. Gilham, J. Liu, D. R. Smith, and S. Westfold. "Planware – Domain-Specific Synthesis of High-Performance Schedulers". In D. F. Redmiles and B. Nuseibeh, (eds.), *Proc. 13th Intl. Conf. Automated Software Engineering*, pp. 270–280. IEEE Comp. Soc. Press, 1998.

[CL+00] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. "A certifying compiler for Java". In *Proc. ACM Conf. Programming Language Design and Implementation 2000*, pp. 95–107. ACM Press, 2000. Published as SIGPLAN Notices 35(5).

[DF03] E. Denney and B. Fischer. "Correctness of Source-Level Safety Policies". In K. Araki, S. Gnesi, and D. Mandrioli, (eds.), *Proc. FM 2003: Formal Methods*, *Lect. Notes Comp. Sci.* **2805**, pp. 894–913. Springer, 2003.

[DFS05] E. Denney, B. Fischer, and J. Schumann. "An Empirical Evaluation of Automated Theorem Provers in Software Certification". *International Journal of AI Tools*, 2005. To appear.

[FH+03] B. Fischer, A. Hajian, K. Knuth, and J. Schumann. "Automatic Derivation of Statistical Data Analysis Algorithms: Planetary Nebulae and Beyond". In G. Erickson and Y. Zhai, (eds.), *Proc. 23rd Intl. Workshop on Bayesian Inference and Maximum Entropy Methods in Science and Engineering*, pp. 276–291. American Institute of Physics, 2003.

[FL$^+$02] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. "Extended static checking for Java". In L. J. Hendren, (ed.), *Proc. ACM Conf. Programming Language Design and Implementation 2002*, pp. 234–245. ACM Press, 2002. Published as SIGPLAN Notices 37(5).

[FS03] B. Fischer and J. Schumann. "AutoBayes: A System for Generating Data Analysis Programs from Statistical Models". *J. Functional Programming*, **13**(3):483–508, 2003.

[Kre98] C. Kreitz. "Program Synthesis". In W. Bibel and P. H. Schmitt, (eds.), *Automated Deduction — A Basis for Applications*, pp. 105–134. Kluwer, 1998.

[LPR01] M. Lowry, T. Pressburger, and G. Rosu. "Certifying Domain-Specific Policies". In M. S. Feather and M. Goedicke, (eds.), *Proc. 16th Intl. Conf. Automated Software Engineering*, pp. 118–125. IEEE Comp. Soc. Press, 2001.

[MI$^+$97] M. Moser, O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann, and K. Mayr. "The Model Elimination Provers SETHEO and E-SETHEO". *J. Automated Reasoning*, **18**:237–246, 1997.

[Mit96] J. C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[Nec97] G. C. Necula. "Proof-Carrying Code". In *Proc. 24th ACM Symp. Principles of Programming Languages*, pp. 106–19. ACM Press, 1997.

[NL98] G. C. Necula and P. Lee. "The Design and Implementation of a Certifying Compiler". In K. D. Cooper, (ed.), *Proc. ACM Conf. Programming Language Design and Implementation 1998*, pp. 333–344. ACM Press, 1998. Published as SIGPLAN Notices 33(5).

[RTC92] RTCA Special Committee 167. Software Considerations in Airborne Systems and Equipment Certification. Technical report, RTCA, Inc., December 1992.

[Smi90] D. R. Smith. "KIDS: A Semi-Automatic Program Development System". *IEEE Trans. Software Engineering*, **16**(9):1024–1043, 1990.

[SW$^+$94] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. "Deductive Composition of Astronomical Software from Subroutine Libraries". In A. Bundy, (ed.), *Proc. 12th Intl. Conf. Automated Deduction*, *Lect. Notes Artificial Intelligence* **814**, pp. 341–355. Springer, 1994.

[WB96] V. L. Winter and J. M. Boyle. "Proving Refinement Transformations for Deriving High-Assurance Software". In *Proc. High-Assurance Systems EngineeringWorkshop*, pp. 68–77. IEEE Comp. Soc. Press, 1996.

[WH99] M. Whalen and M. Heimdahl. "On the Requirements of High-Integrity Code Generation". In *Proc. 4th Intl. Symp. High-Assurance Systems Engineering*, pp. 216–226. IEEE Comp. Soc. Press, 1999.

[WS04] J. Whittle and J. Schumann. "Automating the Implementation of Kalman Filter Algorithms". *ACM Transactions on Mathematical Software*, **30**(4):434–453, 2004.

[WSF02] M. Whalen, J. Schumann, and B. Fischer. "Synthesizing Certified Code". In L.-H. Eriksson and P. A. Lindsay, (eds.), *Proc. Intl. Symp. Formal Methods Europe 2002: Formal Methods—Getting IT Right*, *Lect. Notes Comp. Sci.* **2391**, pp. 431–450. Springer, 2002.