AutoBayes/CC — Combining Program Synthesis with Automatic Code Certification — System Description —

Michael Whalen[†], Johann Schumann[‡], and Bernd Fischer[‡]

[†]Department of Computer Science and Engineering
Univ. of Minnesota, Minneapolis, MN 55455 email: whalen@cs.umn.edu
[‡]RIACS / NASA Ames, Moffett Field, CA 94035
email: {schumann|fisch}@email.arc.nasa.gov

1 Introduction

Code certification is a lightweight approach to formally demonstrate software quality. It concentrates on aspects of software quality that can be defined and formalized via properties, e.g., operator safety or memory safety. Its basic idea is to require code producers to provide formal *proofs* that their code satisfies these quality properties. The proofs serve as *certificates* which can be checked independently, by the code consumer or by certification authorities, e.g., the FAA. It is the idea underlying such approaches as proof-carrying code [6].

Code certification can be viewed as a more practical version of traditional Hoare-style program verification. The properties to be verified are fairly simple and regular so that it is often possible to use an automated theorem prover to automatically discharge *all* emerging proof obligations. Usually, however, the programmer must still splice auxiliary annotations (e.g., loop invariants) into the program to facilitate the proofs. For complex properties or larger programs this quickly becomes the limiting factor for the applicability of current certification approaches.

Our work combines code certification with automatic program synthesis [4] which makes it possible to automatically generate both the code and all necessary annotations for fully automatic certification. By *generating* detailed annotations, one of the biggest obstacles for code certification is removed and it becomes possible to automatically check that synthesized programs obey the desired safety properties.

Program synthesis systems are built on the notion of "correctness-by-construction", i.e., generated programs always implement the specifications correctly. Hence, verifying these programs may seem redundant. However, a synthesis system ensures only that code fragments are assembled correctly while the fragments themselves are included in the domain theory and thus not directly verified by the synthesis proof. Our approach can verify properties about the instantiated code fragments, and so provides additional guarantees about the generated code.

2 The AutoBayes/CC System

AutoBayes/CC (Fig. 1) is a code certification extension to the AutoBayes synthesis system, which is used in the statistical data analysis domain[2]. Its input specification is a statistical model, i.e., it describes how the statistical variables are distributed and depend on each other and which parameters have to be estimated for the given task. AutoBayes synthesizes code by exhaustive, layered application of *schemas*. A schema consists of a code fragment with open slots and a set of applicability conditions. The synthesis system fills the slots with code fragments by recursively calling schemas. The conditions constrain how the slots can be filled; they must be proven to hold for the specification model before the schema can be applied. Some of the schemas contain calls to symbolic equation solvers, others contain entire skeletons of statistical or numerical algorithms. By recursively invoking schemas and composing the resulting code fragments, AutoBayes is able to automatically synthesize programs of considerable size and internal complexity (currently up to 1,400 lines of commented C++ code).

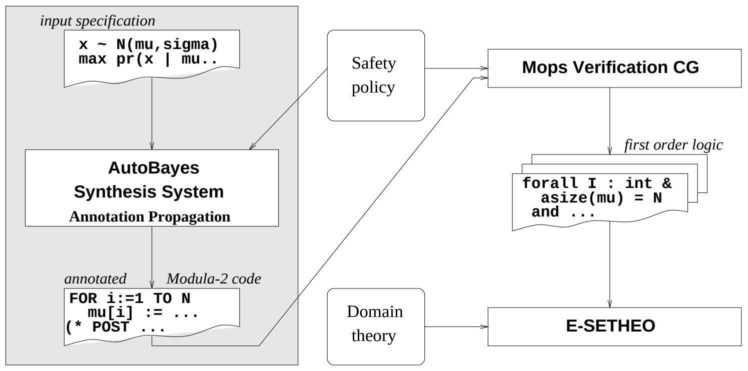

Fig. 1. The AutoBayes/CC system architecture

At the core of the CC-extension lie "certification augmentations" to the AutoBayes-schemas; these augmentations are schematic Hoare-style code annotations that describe how the schema-generated code locally affects properties of interest to our safety policy (currently memory and operator safety). For example, a loop is annotated with a schematic invariant and schematic pre- and postconditions describing how its body changes the variables of the program. During synthesis, the annotations are instantiated in parallel with the original schemas. The domain knowledge encoded in each schema is detailed enough to provide all information required for the instantiation. These annotations are also used to partition the safety proofs into small, automatically provable segments.

Unfortunately, these schema-local annotations are in general insufficient to prove the postconditions at the end of recursively composed fragments—an "inner" loop-invariant may not be aware of proof obligations that are relevant to an "outer" loop-invariant. AutoBayes overcomes this problem by *propagating* any unchanged information through the annotations. Since program synthesis restricts aliasing to few, known places, testing which statements influence which annotations can be accomplished easily without full static analysis of the synthesized program.

As a next step, the synthesized annotated code is processed by a verification condition generator (VCG). Here we use the VCG of the *Modula Proving System* Mops [3], a Hoare-calculus based verification system for a large subset of the programming language Modula-2,[1] including pointers, arrays, and other data structures. Mops uses a subset of VDM-SL as its specification language; this is interpreted here only as syntactic sugar for classical first-order logic.

The proof obligations generated by Mops are then fed (after automatic syntactic transformation and addition of domain axioms) into the automated theorem prover E-Setheo, version csp01 [1]. E-Setheo is a compositional theorem prover for formulas in first-order logic, combining the systems E [8] and Setheo [5]. The individual subsystems are based on the superposition, model elimination, and semantic tree calculi. Depending on syntactic characteristics of the input formula, an optimal schedule for each of the different strategies is selected. Because all of the subsystems work on formulas in clausal normal form (CNF), the first-order formula is converted into CNF using the module Flotter [10].

3 A Certification Example

We illustrate the operation of our system on a standard data analysis task: classify normally (Gaussian) distributed data from a mixture of sources (e.g., photon energy levels in a spectrum). A straightforward 19-line specification is sufficient to describe the problem in domain-specific terms. The synthesized program uses an iterative EM (expectation maximization) algorithm and consists of roughly 380 lines of code, 90 of which are auto-generated comments to explain the code. For details see `http://ase.arc.nasa.gov/schumann/AutoBayesCC` and [2]. The code is annotated to prove division-by-zero and array-bounds safety. With all annotations (including the propagated annotations), the code grows to 2,116 lines—a clear indication that manual annotation is out of question. For an excerpt of the code see Figure 2.

The Mops verification condition generator takes this annotated code file and produces 69 proof tasks in first-order logic. Initially, E-Setheo could solve 65 of the 69 tasks automatically. The remaining four proof tasks were of the general form $Ax \wedge A \wedge B \rightarrow A' \wedge C$ where Ax, A, A', B, C are variable-disjoint first-order formulas. This form is a consequence of the task generation process: Ax represents the domain axioms, A and A' are propagated annotations, and $B \rightarrow C$ is the "proper" proof obligation itself. In order to reduce the formula size, a preprocessing script was used to split each of these proof tasks into two separate tasks, namely $Ax \wedge A \wedge B \rightarrow A'$ and $Ax \wedge A \wedge B \rightarrow C$; these were then processed

[1] We extended AutoBayes to generate the Modula-2 code. Usually, AutoBayes synthesizes C/C++ programs for Matlab and Octave (`http://www.octave.org`).

separately and proven automatically by E-Setheo. After conversion into CNF the formulas had on average 131 clauses (between 112 and 166); roughly half of the clauses were non-Horn. The terms had a syntactically rich structure with an average number of 51 function symbols and 39 constant symbols. Equality and relational operators were the only predicate symbols in the original formula; additional predicate symbols were introduced by Flotter during the conversion into CNF. Despite the size of the formula and their syntactic richness, most of the proofs were relatively short and were basically found by only two of E-Setheo's strategies, namely using the E-prover, and an iterative combination of the E-prover and scheme-SETHEO.

```
1 (*{ assert i=N and j=M and
2  (forall a,b : int & ((0<=a and a<N) and
3    (0<=b and b<M)) => q[a,b]=0.0) }*)
4 (*{ loopinv  0<=k and k<=N-1 and
5  (forall a,b: int &  ((0<=a and a<N) and
6    (0<=b and b<M) => 0<=q[a,b] and
7      q[a,b]<=1.0) }*)
8  FOR k := 0 to N - 1 DO
9     q[k,c[k]] := 1.0;
10 END
11 (*{ post (forall a,b : int &
12  ((0<=a and a<N) and (0<=b and b<M))
13    => 0 <= q[a,b] and q[a,b] <=  1.0) }*)
```

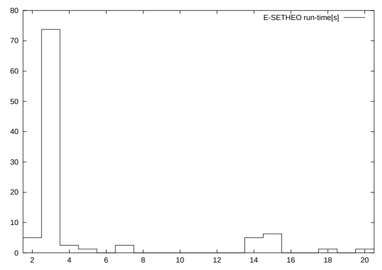

Fig. 2. *Left*, excerpt of annotated code produced by AutoBayes/CC. Annotations are enclosed in `(*{...}*)`. *Right*, distribution of E-Setheo proof times (% solved over runtime in seconds).

Fig. 2 shows the runtime distribution for the proof tasks.[2] Most tasks were solved in about two to three seconds, but some tasks took up to 20 seconds. The smaller second peak visible around 15 seconds is due to a non-optimal schedule. We expect that a re-training of E-Setheo's internal scheduler could help to avoid such long runtimes (cf. [9]). The overall proof time of 323 seconds indicates that our approach is feasible.

In order to compare our approach to certification techniques based on static analysis, we analyzed the equivalent C-version of our example program with the commercial tool PolySpace [7]. PolySpace was capable of declaring most of the code safe with respect to memory/operator safety. However, it could not clear several crucial parts of the code, most notably the nested indexing (`q[k,c[k]]`, see line 9 in Fig. 2) and the initialization of some variables in the main loop. In these cases, certification requires annotation propagation as it is done in our work; Polyspace does not require or support annotations. On the other hand, PolySpace detected a possible integer overflow error of a loop counter in the synthesized code, something that our safety policy does not (yet) check. The runtime of PolySpace for this example (about one hour of wall-clock time on the

[2] All runtimes have been obtained with a total CPU-time limit of 120 seconds on a 1000 MHz SunBlade workstation. Due to the internal scheduling of E-Setheo, substantially different runtimes can result if this limit is changed.

same 1000MHz SunBlade) demonstrates that our approach can be competitive to commercial tools.

4 Conclusions

In this paper, we have described AutoBayes/CC, a novel combination of automated program synthesis and automated program verification. Our idea is to use the knowledge of the domain which is formalized in the program synthesis system to generate the program together with the necessary detailed formal annotations required for a fully automatic safety proof. The underlying approach is general and we expect it to be applicable to other code-generation systems as well. The major benefit of this combination of program synthesis and program verification is obviously the additional verification of important quality aspects of the synthesized code which comes at no cost for the user.

AutoBayes/CC is still work in progress; currently, the certification extension covers only those parts of the domain theory required to generate EM-variants. However, we see no fundamental obstacles in extending the approach to the entire (still growing) domain theory. Also, the safety policy is still hard-coded in the way the annotations are generated within the synthesis schemas. We will work on ways to explicitly represent safety policies (e.g., using higher-order formulations) and use this to tailor the annotation generation in AutoBayes/CC. We also plan to implement a preprocessing and simplification component which can substantially reduce size and complexity of the proof tasks.

References

[1] CASC-JC Theorem Proving Competition. `www.cs.miams.edu/~tptp/CASC/JC`, 2001.

[2] B. Fischer and J. Schumann. AutoBayes: A System for Generating Data Analysis Programs from Statistical Models. *JFP*, to appear 2002. Preprint available at `http://ase.arc.nasa.gov/people/fischer/papers.html`.

[3] T. Kaiser, B. Fischer, and W. Struckmann. "MOPS: Verifying Modula-2 programs specified in VDM-SL". *Proc. 4th Workshop Tools for System Design and Verification*, pp. 163–167, 2000.

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

[5] 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.

[6] G. C. Necula. "Proof-Carrying Code". *Proc. 24th POPL*, pp. 106–119. 1997.

[7] PolySpace Technologies. `www.polyspace.com`, 2002.

[8] S. Schulz. "System Abstract: E 0.3". *Proc. 16th CADE*, *LNAI* 1421, pp. 297–301. 1999.

[9] G. Stenz and A. Wolf. "E-SETHEO: Design Configuration and Use of a Parallel Theorem Prover". *Proc. 12th Australian Joint Conf. Artificial Intelligence*, *LNAI* 1747, pp. 231–243. 1999.

[10] C. Weidenbach, B. Gaede, and G. Rock. "Spass and Flotter version 0.42". *Proc. 13th CADE*, *LNAI* 1104, pp. 141–145. 1996.