Certification Support for Automatically Generated Programs

Johann Schumann[†], Bernd Fischer[†], Mike Whalen[‡], Jon Whittle[⋆]
[†]RIACS / NASA Ames, {schumann,fisch}@email.arc.nasa.gov
[‡]Dept. of CS, Univ. of Minnesota, Minneapolis, whalen@cs.umn.edu
[⋆]QSS / NASA Ames, jonathw@email.arc.nasa.gov

Abstract

Although autocoding techniques promise large gains in software development productivity, their "real-world" application has been limited, particularly in safety-critical domains. Often, the major impediment is the missing trustworthiness of these systems: demonstrating—let alone formally certifying—the trustworthiness of automatic code generators is extremely difficult due to their complexity and size.

We develop an alternative product-oriented certification approach which is based on five principles: (1) trustworthiness of the generator is reduced to the safety of each individual generated program; (2) program safety is defined as adherence to an explicitly formulated safety policy; (3) the safety policy is formalized by a collection of logical program properties; (4) Hoare-style program verification is used to show that each generated program satisfies the required properties; (5) the code generator itself is extended to automatically produce the code annotations required for verification. The approach is feasible because the code generator has full knowledge about the program under construction and about the properties to be verified. It can thus generate all auxiliary code annotations a theorem prover needs to discharge all emerging verification obligations fully automatically.

Here we report how this approach is used in a certification extension for AUTOBAYES*, an automatic program synthesis system which generates data analysis programs (e.g., for clustering and time-series analysis) from declarative specifications. In particular, we describe how a variable-initialization-before-use safety policy can be encoded and certified.*

1 Introduction

Autocoding techniques (also called code-generation or automatic program synthesis) are concerned with the translation of high-level specifications into code. These techniques can improve productivity by allowing developers to specify software behavior at a high level of abstraction, leaving the code generator to manage the implementation details. If correct, these techniques also prevent errors from being introduced in the implementation process, thus improving software quality. Autocoding is therefore an enticing approach for safety-critical or security-critical application domains, e.g., spacecraft navigation and control. However, in these domains, it faces a crucial dilemma: can users really trust the code generators? Demonstrating the trustworthiness of a code generator is extremely difficult due to its complexity and size, so the use of autocoding approaches in safety-critical domains has been minimal. In this paper, we describe an alternative approach which demonstrates trustworthiness for each generated program (i.e., the product) separately, rather than for the entire generator system (i.e., the process).

Trustworthiness is a multi-faceted software quality feature; typically, it involves questions like "Can the vendor be trusted?", "Can the user be trusted?", or, obviously, "Can the code itself be trusted to behave as expected?" In this paper we focus on the last aspect; more precisely, we focus on the certification of code with respect to certain properties that are usually expected to hold for trustworthy code. Typical properties, which our approach can handle, are array bounds safety, operator safety, and variable-initialization-before-use. Violations of these properties (usually "buffer overflow", which is an array-bounds violation) have caused several safety- and security-critical incidents, including many Internet worms.

Due to the complexity and size of the generated programs, however, neither testing nor code inspections can realistically be used to demonstrate that the selected properties hold. Our approach thus uses *formal code certification*. Our basic idea is to extend the code generator such that it provides formal *proofs* that the generated code satisfies the selected properties. These proofs serve as *certificates* which can be checked independently, by the code consumer or by a certification authority, for example the FAA. Code certification is based on the same technology as Hoare-style program verification; in particular, it also uses code annota-

tions to formalize the properties. The difference, however, is in the details: the properties are much simpler and more regular than full behavioral specifications. Both aspects are crucial. Since the properties are simple and regular, the annotations can be derived schematically from a public *safety policy* and automatically inserted into the generated code. Proving these properties, given annotated programs, is then straightforward, and can be performed by an automated theorem prover (ATP).

Our approach is similar to the concept of proof carrying code (PCC) [16]. It also relies on a small, simple kernel of trusted components. These components, the *safety policy*, *verification condition generator* (VCG), and the *proof checker*, are very simple and can be verified using standard software development techniques. Errors or malicious tampering in the complex parts of the software (i.e., the code generator and theorem prover) will cause the proof process to fail at some step. Surprisingly, this holds even though we use the code generator itself—and not the programmer, as in PCC—to produce the auxiliary annotations (e.g., loop invariants) which are required to make the proofs possible; in particular, a wrong (i.e., too strong) loop invariant will just cause the proof process to fail at a later step. Thus, if we can prove several safety properties about the generated code, we can gain a very high level of trust that this code will, in fact, run safely.

This paper elaborates our approach in [22, 21] where we described an extension of the program synthesis system AUTOBAYES which is able to certify generated code with respect to operator safety and array bounds safety. In this paper we have a systematic look at certifiable program properties and show an initial taxonomy of these properties (cf. Section 2). Section 3 gives the necessary background information on proof carrying code and program synthesis, and describes the extended architecture of AUTOBAYES. Section 4 then focuses on the formalization of the safety policies, in particular variable-initialization-before-use, using an extended set of Hoare rules. Our approach allows us to easily customize a safety policy in order to enforce additional constraints, e.g., programming standards. In Section 5, we describe how the code annotations are generated within the synthesis system, and how the proof obligations are produced by the verification generator and processed by the automated theorem prover. Section 6 relates our work to other approaches in verification, certification, and generation of trustworthy code; in Section 7, we summarize and sketch current and future work.

2 Property Verification

Traditionally, program verification has focused on showing the functional equivalence of (full) specification and implementation. However, this verification style is extremely demanding, because of the involved specification and proof efforts, respectively. Furthermore, many aspects of trustworthiness are usually not expressed in the specification and thus not demonstrated explicitly. More recent approaches thus concentrate on showing specific properties that are important for software safety and security.

While many mechanisms and tools for verifying program properties have been published, especially for distributed systems (e.g., model checking), relatively little attention has been paid to the properties themselves. The related work in this area is usually concerned with computer security [19]; we are interested in all "useful" and "important" properties. To help guide our research, we have created an initial taxonomy of verifiable properties of programs. A first attempt is shown in Figure 1.

Safety properties prevent the program from performing illegal or nonsensical operations, such as access to unavailable memory addresses or division by zero. Within this category, we further subdivide into five different aspects of safety. *Memory safety properties* assert that all memory accesses involving arrays and pointers are within their assigned bounds. *Type safety properties* assert that a program is "well typed" according to a type system defined for the language. This type system may correspond to the standard type system for the language, or may enforce additional type checking obligations, such as ensuring that all variables representing physical quantities have correct and compatible units and dimensions (cf. [14]). *Numeric safety properties* assert that programs will perform arithmetic operations correctly. Potential errors include (1) using partial operators, like divide or square root, with arguments outside their defined domain, (2) performing computations that yield results larger (overflow) or smaller (underflow) than are representable on the computer, and (3) performing floating point operations which cause an unacceptable loss of precision. *Exception handling properties* ensure that all exceptions that can be thrown within a program are handled within the program. *Environment compatibility properties* ensure that the program is compatible with its target environment. Compatibility constraints specify hardware, operating systems, and libraries necessary for safe execution. Parameter conventions define constraints on program communication and invocation.

Resource limit properties check that the required resources for a computation are within some bound. *Liveness/progress properties* are used to show that the program will eventually perform some required activity, or will not be permanently blocked waiting for resources. *Security properties* prevent a program from accidental or malicious tampering with the environment and from being modified by an attacker.

Trustworthiness of a program cannot be related to just one property; only if the software together its environment

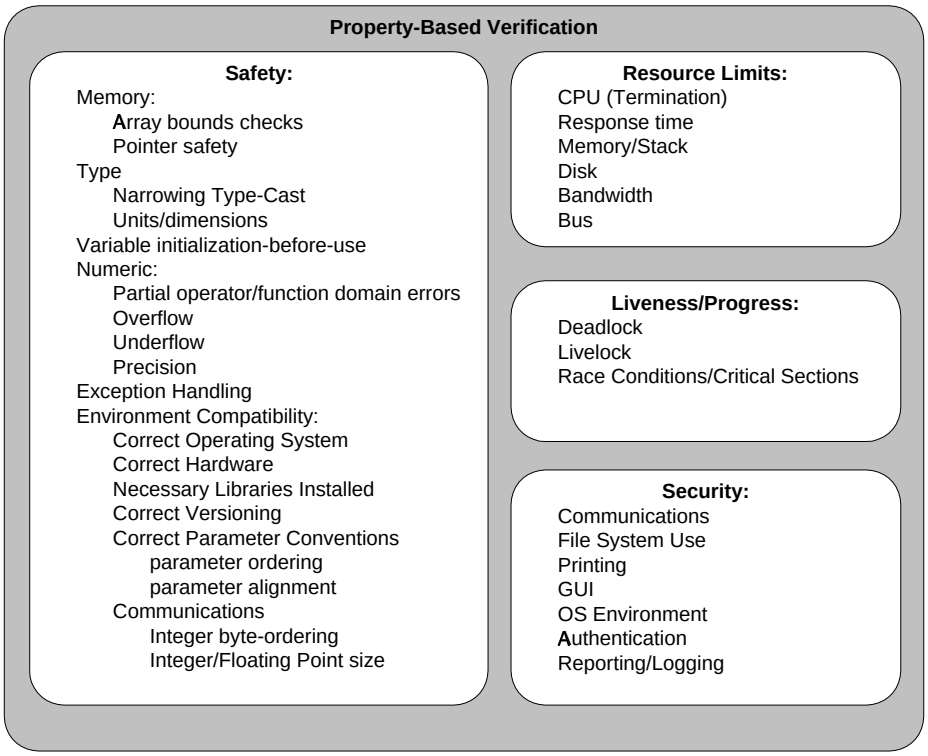

Figure 1. Overview of verification properties

fulfills a set of properties it can be considered trustworthy. Clearly, there is overlap between the properties in the different categories; for example, many security flaws are due to safety violations (e.g., memory safety violations in form of buffer overflows). We plan to extend and clarify this taxonomy in future work. For this project, we are interested in non-trivial properties that are amenable to automatic verification. Therefore, in this paper we concentrate our investigations on a particular safety property: variable definition before use. Two other properties, operator safety and array-bounds safety are already discussed in [22, 23].

3 Background

3.1 Proof-Carrying Code

PCC [16, 1] is a certification approach especially for mobile code. Many distributed systems (e.g., browsers, cellular phones) allow the user to download executable code and run it on the local machine. If the origin of this code is unknown, or the source is not trustworthy, this poses a considerable risk: the code may not be compatible with the current system status, or the code can destroy critical data.

The PCC concept and system architecture (Figure 2) have been developed to address the problem of showing certain properties (i.e., the safety policy) efficiently at the time when the software is downloaded. The developer of the software annotates the program which is then compiled into object-code using a certifying compiler, e.g., Touchstone [3]. Such a compiler carries over the source code annotations to the object code level. A VCG processes the annotated code together with a public safety policy. The VCG produces a large number of proof obligations. If all of them are proven, the safety policy holds for this program. However, since these activities are performed by the producer, the provided proofs are not necessarily trustworthy. Therefore, the annotated code and a compressed copy of the proofs are packaged together and sent to the user. The user reconstructs the proof obligations and uses a proof checker to ensure that the conditions match up with the proofs as delivered with the software. Both, the VCG and the proof checker, need to be trusted in this approach. However, since a proof checker is much simpler in its internal structure than a prover, it is simpler to design and implement in a correct and trustworthy manner. Furthermore, checking a proof is very efficient, in stark contrast to finding the proof in the first place—which is usually a very complex and time-consuming process.

A number of PCC-approaches have been developed, particularly focusing on the compact and efficient representation of proofs (e.g., using LCF [16] or HOL [1]). However, all of these approaches are in practice restricted to very sim-

ple properties. More intricate properties require the producer of the program to provide elaborate annotations and to carry out complicated formal proofs manually.

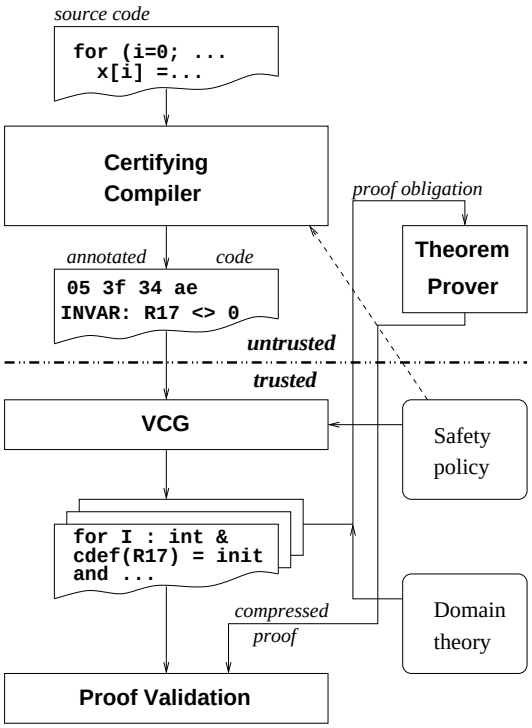

Figure 2. Typical architecture for proof carrying code. Trusted components are below the dot-dashed line.

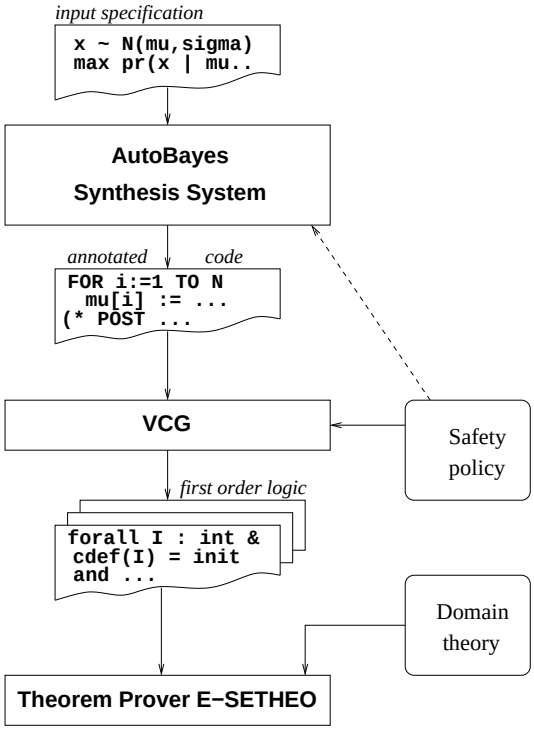

Figure 3. AutoBayes system architecture, extended for code certification

3.2 Program Synthesis and AutoBayes

Automated program synthesis aims at automatically constructing executable programs from high-level specifications. It is usually based on mathematical logic, although a variety of different approaches exist [13]. Here, we will focus on a specific approach, schema-based program synthesis, and a specific system, AUTOBAYES. AUTOBAYES [6] generates complex data analysis programs from compact specifications in the form of statistical models. It has been applied to a number of domains, including clustering, change detection, sensor modeling, and software reliability modeling, and has been used to generate programs with up to 1500 lines of C++ code.

AUTOBAYES synthesizes code by repeated application of *schemas*. A schema consists of a program fragment with open slots and a set of applicability conditions. The slots are filled in with code pieces by the synthesis engine calling schemas recursively. The conditions constrain how the slots can be filled; they must be proven to hold in the given specification 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.

Figure 4 below shows in stylized Prolog-notation a slightly simplified schema which is selected when a function needs to be maximized. It synthesizes a code fragment C which calculates the maximum w.r.t. a single variable X for a symbolically given function F. The applicability of this schema is restricted to cases where a first derivative of F exists. The schema first tries to compute the maximum symbolically by solving the equation $\partial \mathsf{F}/\partial \mathsf{X} = 0$ for X. If that succeeds, it returns a single assignment. Otherwise, an iterative numerical optimization routine must be synthesized in order to solve the given problem. Such an algorithm consists of three code segments: finding a start value x_0, calculation of the search direction p, and the step length λ. Then, starting with x_0, the maximum is sought by iteratively approaching the maximum: $x_{k+1} = x_k + \lambda_k p_k$ (for details see [9, 17]). Our schema assembles this algorithm by re-

cursive calls to schemas to obtain code fragments `CInit`, `CSteplength`, and `CStepdir` for initialization, calculation of the step length, and step direction, respectively. Instantiation of code fragments in the algorithm skeleton is denoted by `<...>`.

```
schema(max F wrt X, C) :-
  exists(first-derivative(F)),
  symbolic_solve(d(F, X) == 0, Solution),
  if (Solution != not_found)
    C = "<X> := <Solution>";
  else {
    schema(getStartValue(F,X), CInit);
    schema(getStepsize(F,X),CSteplength);
    schema(getStepdir(F,X),CStepdir);
     C = "{ <CInit>;
       while(converging(<X>))
         <X> := <X> +
                <CSteplength>*<CStepdir>; }";
    }.
```

Figure 4. Synthesis Schema (Fragment)

While we cannot present details of the synthesis process here, we want to emphasize that the code is assembled from building blocks which are obtained by symbolic computation or schema instantiation. The schemas clearly lay out the domain knowledge and important design decisions. As we will see later on, they can be extended in such a way that the annotations required for the certification are also generated automatically.

3.3 AutoBayes/CC

The architecture of our certifying synthesis system (see Figure 3) is similar to the typical proof-carrying code architecture shown in Figure 2. However, since we are currently not dealing with proof validation aspects, we only have three major building blocks: the synthesis system AUTOBAYES (which replaces the certifying compiler), the verification condition generator, and the automated theorem prover.

The system's input is a specification of a statistical model. This specification need not be modified for certification—the process is thus completely transparent to the user. AUTOBAYES then attempts to synthesize code using the schemas described above. These schemas are extended appropriately to support the automatic generation of code annotations. AUTOBAYES produces Modula-2 code[1] which carries the annotations as comments. Annotations and code are then processed by the verification condition generator MOPS[11]. Its output is a set of proof obligations in first order predicate logic which must be proven to show the desired properties. In order to do so, a domain theory in form of a set of axioms must be added to the formulas. Finally, these extended proof obligations are fed into the automated theorem prover E-SETHEO[2].

[1]Since MOPS works on Modula-2, we extended AUTOBAYES to generate Modula-2 code. Usually, AUTOBAYES synthesizes C++/C programs for the Octave and Matlab environments.

4 Safety Policies

The first step in our approach is to define precisely what constitutes our properties by formulating them as predicates within a logic. Then, we must define some mechanism to transform a program into a series of verification conditions that are valid if and only if the safety properties are satisfied.

Hoare rules [25] form the foundation of our approach. They are triples of the form $\{P\}\ C\ \{Q\}$ where C is a statement in an imperative programming language, and P and Q are predicates. The statement acts as a *predicate transformer*, that is, it describes how (the state described by) predicate P is transformed into predicate Q by the execution of C. Our idea is to add explicit constraints to the predicates in the rules to ensure that each statement fulfills the required properties. We use a predicate $\mathit{SafeExpr}_{\mathcal{P}}(E)$ which is true iff property $\mathcal{P}$ holds for the expression E. For the most part, these modifications are in the form of strengthened preconditions. In the sections below, we will give a formal definition of this predicate for our selected safety policy.

4.1 Variable Initialization Before Use

At first glance, an initialization-before-use safety policy may seem unnecessary for two reasons: First, modern compilers can detect many violations of this property. However, more intricate cases (e.g., within loops and arrays) are not handled by the usual compilers. Second, for full functional equivalence proofs, uninitialized variables will in many cases cause proof obligations to be unprovable.[2] However, when proof obligations are relatively weak, as it is for safety policies, uninitialized variables do not necessarily cause proof obligations to fail but can still cause a program to yield incorrect results. Therefore, an *explicit* initialization-before-use property is an important aspect of any safety policy.

To formalize the policy, we must first add the notion of variable initializations, rather than simply variable values, to the logic. This could be accomplished by adding an explicit *bottom*-value to the domains, as usual in denotational semantics. However, this complicates reasoning about the proof obligations unnecessarily. We thus use *shadow vari-*

[2]If the obligations remain provable, the use of an initialized variable is irrelevant to the specified functionality and could be removed.

Scalar Assign $\{P[e/x, \text{INIT}/x_{init}] \wedge \mathit{SafeExpr}_I(e)\}\ x := e\ \{P\}$

Array Assign
$$\left\{\begin{array}{l} P[x\{(e_0,\ldots,e_n) \rightarrow e\}, \\ \quad x_{init}\{(e_0,\ldots,e_n) \rightarrow \text{INIT}\}] \\ \wedge\ \mathit{SafeExpr}_I(e) \\ \wedge\ \mathit{SafeExpr}_I(e_0) \\ \quad \ldots \\ \wedge\ \mathit{SafeExpr}_I(e_n) \end{array}\right\} x[e_0,\ldots,e_n] := e\ \{P\}$$

Conditional
$$\frac{\{P \wedge b \wedge \mathit{SafeExpr}_I(b)\}\ s\ \{Q\} \quad (P \wedge \neg b \wedge \mathit{SafeExpr}_I(b) \implies Q)}{\{P \wedge \mathit{SafeExpr}_I(b)\}\ \texttt{if}\ b\ \texttt{then}\ s\ \{Q\}}$$

While Loop
$$\frac{\{P \wedge b \wedge \mathit{SafeExpr}_I(b)\}\ c\ \{P \wedge \mathit{SafeExpr}_I(b)\}}{\{P \wedge \mathit{SafeExpr}_I(b)\}\ \texttt{while}\ b\ \texttt{do}\ c\ \{P \wedge \neg b \wedge \mathit{SafeExpr}_I(b)\}}$$

For Loop
$$\frac{\left\{\begin{array}{l} P[\text{INIT}/x_{init}] \wedge \\ \quad e_0 \leq x \leq e_1 \wedge \\ \quad \mathit{SafeExpr}_I(e_0) \wedge \\ \quad \mathit{SafeExpr}_I(e_1) \end{array}\right\}\ s\ \left\{\begin{array}{l} P[(x+1)/x] \wedge \\ \quad \mathit{SafeExpr}(e_0) \wedge \\ \quad \mathit{SafeExpr}(e_1) \end{array}\right\}}{\left\{\begin{array}{l} P[e_0/x, \text{INIT}/x_{init}] \wedge \\ \quad \mathit{SafeExpr}_I(e_0) \wedge \\ \quad e_0 \leq e_1 \wedge \mathit{SafeExpr}_I(e_1) \end{array}\right\}\ \texttt{for}\ x := e_0\ \texttt{to}\ e_1\ \texttt{do}\ s\ \left\{\begin{array}{l} P[(e_1+1)/x] \wedge \\ \quad \mathit{SafeExpr}_I(e_0) \wedge \\ \quad \mathit{SafeExpr}_I(e_1) \end{array}\right\}}$$

Sequence
$$\frac{\{P\}\ s_0\ \{R\} \quad \{R\}\ s_1\ \{Q\}}{\{P\}\ s_0; s_1\ \{Q\}}$$

Conseq. Rule
$$\frac{P' \implies P \quad \{P\}\ s\ \{Q\} \quad Q \implies Q'}{\{P'\}\ s\ \{Q'\}}$$

Figure 5. Hoare rules with init-before-use safety policy extensions

ables in the specification that keep track of the status of program variables. In our case, for each variable x, we associate a shadow variable x_{init}, where x_{init} can either have the value INIT or UNINIT. Whenever the variable is assigned a value, the shadow variable is set to INIT; whenever the variable is used, an obligation is added to ensure that the shadow variable is equal to INIT. The shadow variables are pure specification-level variables, i.e., they are not accessible by the program.

More concretely, to check safety, we define a function *VariableRefs*, that returns all variable references (including array subscripts) within an expression. Then, a safe expression, with respect to our initialization policy can be defined:

$$\mathit{SafeExpr}_I(E) \equiv \forall v \in \mathit{VariableRefs}(E) : v_{init} = \text{INIT}$$

The $\mathit{SafeExpr}_I$ predicate is higher-order, as it quantifies over expressions in the program syntax. However, since $\mathit{VariableRefs}(E)$ yields a finite set, we can expand these quantified predicates over variables and expressions into a sequence of first-order predicates. For example, given the statement:

$$q[k, c[k]] := 1.0;$$

Then, $\mathit{SafeExpr}_I(q[k, c[k]])$ yields the following safety predicate, once expanded:

$$k_{init} = \text{INIT} \ \wedge\ c_{init}[k] = \text{INIT} \ \wedge\ q_{init}[k, c[k]] = \text{INIT}.$$

4.2 Modified Hoare-Rules

Figure 5 specifies the entire safety policy via modified Hoare-rules. The first rule describes scalar assignments, and is the same as the standard Hoare assignment rule, except that it has a strengthened precondition that checks whether all variables referenced within the assignment expression have been initialized. The second rule describes assignment of array cells. Unlike scalar assignment, array cell assignment cannot be handled by simple substitution, because of the possibility of aliasing of array cells. Instead, we think of the array as describing a mapping function from cells to values. An assignment to a cell is an update of

the mapping function, written as $x\{(e_0, e_1, \ldots, e_n) \rightarrow e\}$. This approach is the standard extension of the Hoare calculus to handle arrays and is described fully in [15]. We strengthen the precondition of this rule to ensure that both the subscript expressions in the left-hand side and the assignment expression are safe.

The next three rules describe conditional and loop statements. They are the same as the standard Hoare rules, with strengthened preconditions to show that their expressions are safe. Finally, we define the standard Hoare rule of consequence, which states that we can always legally strengthen the precondition or weaken the postcondition of a statement. Soundness of all rules is obvious.

4.3 Customizing the Safety Policy

Coding standards are a traditional way to increase the trustworthiness of software systems. They prohibit the use of certain legal but error-prone coding practices. Most software processes for safety-critical applications thus require that the developed software follows a given coding standard. Obviously, automatically generated code must also adhere to these standards. By formalizing the coding standards as safety policies and by extending the Hoare-rules described above, we can in fact demonstrate formally that the generated code follows customized coding standards.

As an example, we will customize our policy such that it supports the following coding standard: *"Index variables for* `for`*-loops shall not be used outside their enclosing loops."* This standard prohibits situations where the loop index variable is abused to force a premature loop exit and its value is later used to check whether the loop was aborted or finished properly (Figure 6A).

(A)
```
FOR i := 1 TO N DO
  <...>
  IF <...> THEN
    i := N + 2 (* abort *)
  END
END
IF i = N + 2
THEN <...> (* aborted? *)
ELSE <...> (* terminated? *)
END
```

(B)
```
FOR i := 1 TO N DO
  <...>
  FOR i := 1 TO M DO
    <...>
  END
  <...>
END
```

Figure 6. Violations of the coding standard

The code example in Figure 6B shows another effect of our extended safety policy. Here, the two nested loops erroneously use the same index variable which can lead to unintended program behavior. Such improperly constructed loops can easily arise if the code generator is not implemented carefully. However, our extended safety policy catches this situation.

The policy is implemented by adding two more values to the domain of our shadow variables: LOOP and STALE. These values represent the state of a loop-index variable within the loop and following the loop, respectively. We then modify the definition of *SafeExpr* to allow expressions inside a loop to use loop variables:

$$\mathit{SafeExpr}_I(E) \equiv \forall v \in \mathit{VariableRefs}(E) : (v_{init} = \text{INIT} \vee v_{init} = \text{LOOP})$$

Next, we need to modify both the assignment rules and the for-loop rule in order to distinguish between loop variables and variables assigned by other means. These changes are shown in Figure 7. For scalar and array assignments, we can only assign variables which have not been used as a loop index (i.e., INIT or UNINIT). In the for-loop rule, we add a constraint that our loop variable has not previously been assigned a value. Within the loop, we assert that the variable is being used as a loop counter ($P[\text{LOOP}/x_{init}]$), and outside the loop, that the variable is "stale", and cannot be used ($P[\text{STALE}/x_{init}]$).

5 Processing Annotated Code

5.1 The Verification Condition Generator

In the typical proof-carrying code architecture as shown in Figure 2 the safety policy is a separate component. In practice, however, it is hardcoded into the verification condition generator component of the certifying compiler. In our current implementation, all required annotations are generated so that in principle any VCG can be used. For the experiments, we used the VCG of the *Modula Proving System* MOPS [11]. MOPS is a Hoare-calculus based verification system for a large subset of the programming language Modula-2. It uses a subset of VDM-SL as its specification language; this is interpreted here only as syntactic sugar for classical first-order logic.

5.2 Annotations and their Propagation

Annotating the large programs created by AUTOBAYES requires careful attention to detail and many annotations. There are potentially dozens of loops requiring an invariant, and nesting of loops and if-statements can make it difficult to determine what is necessary to completely annotate a statement. For this reason, we split the task of creating the statement annotations into two parts: creating *local* annotations during the run of AUTOBAYES, i.e., the

Scalar Assign

$$\left\{\begin{array}{l} P[e/x,\ \text{INIT}/x_{init}] \wedge \mathit{SafeExpr}_I(e) \\ \quad \wedge (x_{init} = \text{INIT} \vee x_{init} = \text{UNINIT}) \end{array}\right\}\ x := e\ \{P\}$$

Array Assign

$$\left\{\begin{array}{l} P[x\{(e_0,\ldots,e_n) \to e\}, \\ \quad x_{init}\{(e_0,\ldots,e_n) \to \text{INIT}\}] \\ \wedge(x\{(e_0,\ldots,e_n) \to e\} = \text{INIT}\vee \\ \quad x\{(e_0,\ldots,e_n) \to e\} = \text{UNINIT}) \\ \wedge\ \mathit{SafeExpr}_I(e) \\ \wedge\ \mathit{SafeExpr}_I(e_0) \\ \quad \ldots \\ \wedge\ \mathit{SafeExpr}_I(e_n) \end{array}\right\}\ x[e_0,\ldots,e_n] := e\ \{P\}$$

For Loop

$$\frac{\left\{\begin{array}{l} P[\text{LOOP}/x_{init}] \wedge \\ \quad e_0 \le x \le e_1 \wedge \\ \quad \mathit{SafeExpr}_I(e_0) \wedge \\ \quad \mathit{SafeExpr}_I(e_1) \end{array}\right\}\ s\ \left\{\begin{array}{l} P[(e_1+1)/x] \wedge \\ \quad \mathit{SafeExpr}(e_0) \wedge \\ \quad \mathit{SafeExpr}(e_1) \end{array}\right\}}{\left\{\begin{array}{l} P[e_0/x,\ \text{STALE}/x_{init}] \wedge \\ \quad (x_{init} = \text{UNINIT} \vee x_{init} = \text{INIT}) \wedge \\ \quad \mathit{SafeExpr}_I(e_0) \wedge \\ \quad e_0 \le e_1 \wedge \mathit{SafeExpr}_I(e_1) \end{array}\right\}\ \texttt{for}\ x := e_0\ \texttt{to}\ e_1\ \texttt{do}\ s\ \left\{\begin{array}{l} P \wedge \\ \quad \mathit{SafeExpr}_I(e_0) \wedge \\ \quad \mathit{SafeExpr}_I(e_1) \end{array}\right\}}$$

Figure 7. Modified Hoare rules for coding standards safety policy

proper synthesis process, and then *propagating* the annotations through the code. The schema-guided synthesis mechanism of AUTOBAYES makes it easy to produce annotations local to the current statement, as the annotations are tightly coupled to the individual schema. The local annotations for a statement describe the changes in variables made by that statement, without needing to describe all of the global information that may later be necessary for proofs. Then, the propagation algorithm (see [22] for details) pushes the annotations through the program until they are needed.

Figure 8 shows a small piece of annotated code which initializes a matrix `q` for an iterative statistical algorithm. A value of one is set to exactly one element in each row `pv57`; the column of this element is given by the value of `c[pv57]` (for details see [6, 23]). All annotations are written as Modula-2 comments enclosed in `(*{...}*)`. Pre- and post-conditions start with the keywords `pre` and `post`, respectively, loop invariants with a `loopinv`, and additional assertions with an `assert`.

5.3 The Automated Prover

In order to process the generated proof obligations, we are using the automated theorem prover E-SETHEO, version csp01 [2]. E-SETHEO is a high-performance theorem prover for formulas in first order logic. Out of the 70 generated proof tasks of our example, E-SETHEO could solve 68 automatically with a run-time limit of 60 seconds on a 1000 MHz. SunBlade workstation. The remaining two proof tasks required some preprocessing before they could be proven automatically. These preprocessing steps have been done manually for this experiment; for future versions, we will automate these additional steps. Most of the proof tasks could be solved in about two seconds, but several tasks took up to 20 seconds CPU time. The overall runtime of the prover for all proof tasks was roughly 400 seconds, demonstrating the practical feasibility of our approach.

6 Related Work

The approach most closely related to ours is proof-carrying code which has already been discussed in Section 3.1. However, due to its focus on mobile code, PCC covers many aspects we are not yet interested in, e.g., efficient proof representation and proof checking. It also works on the level of object code or typed intermediate languages (e.g., Flint [20]) and is thus complementary to our approach. Certifying compilers as Touchstone [3] or Cyclone [10] could consequently be used to show that the safety policy established on the source code level is not compromised by the compilation step.

Our methodology uses domain knowledge, built into the synthesis system to automatically generate all the required annotations. In contrast to this, many reverse engineering

```
(*{loopinv
  (forall j: int & (0 <= j and j < N) =>
    (c_init(j) = init)) and
  0 <= pv57 and pv57 <= N and
  pv57_init = init and N_init = init and
  C_init = init and
  (forall a, b: int & (0 <= a and a < N
    and 0 <= b and b < C) =>
       q_init(a, b) = init) }*)
FOR pv57 := 0 TO N - 1 DO
  (*{assert
     (forall j: int & (0 <= j and j < N) =>
       (c_init(j) = init)) and
     (forall a, b: int & (0 <= a and a < N
        and 0 <= b and b < C) =>
          q_init(a, b) = init) and
     N_init = init and C_init = init and
     0 <= pv57 and pv57 < N and
     pv57_init = init and
     c_init(pv57) = init}*);
  q[pv57 ][ c[pv57]] := 1;
END;
(*{assert
  (forall j: int & (0 <= j and j < N) =>
    (c_init(j) = init)) and
    (forall a,b: int & (0 <= a and a < N
      and 0 <= b and b < C) =>
        q_init(a, b) = init) and
    N_init = init and C_init = init }*);
```

Figure 8. Excerpt of synthesized code with annotations. Actual Modula-2 statements are underlined.

approaches try to recover formal specifications and annotations from pure code. Gannod and Cheng [8] use a strongest postcondition predicate transformer to support different reverse engineering tasks but their approach still requires additional manual annotations (e.g., loop invariants). Ernst et al. [4] try to infer such invariants dynamically, using a generate-and-test approach: potential invariants are generated from a set of patterns and checked against previously collected run-time trace information. However, the inferred predicates are not proven to be actually invariant so that the approach is not suitable for certification purposes. Flanagan and Leino [7] describe a similar system, Houdini, to support their ESC/Java verification system. Houdini also uses a generate-and-test approach but the test phase relies on ESC/Java to prove the invariants. However, Houdini does not use domain knowledge in the generate phase and is thus restricted in the kind of invariants it can recover.

Obviously, our research is also related to standard program verification. However, program verification concentrates on showing full functional equivalence rather than property verification. Lowry et al. [14] present an approach for certifying domain-specific properties which is based on abstract interpretation. They check programs for *frame safety*, an extended type safety property. Other safety properties can also be encoded in extended type systems and then checked via (extended) type inference algorithms. Such approaches have been used to show, for example, unit and dimensional safety [18, 12] and memory safety [26]. However, these approaches usually also require additional annotations, e.g., type declarations. Moreover, most of them are restricted to a specific safety policy and thus less general than proof-based certification approaches.

7 Conclusions

In this paper, we have described a novel combination of automated program synthesis and automated program verification with the aim to increase trustworthiness of automatically generated code. Our basic idea is to generate the program *together* with detailed formal annotations which are required for a fully automatic proof of safety properties. This approach is facilitated by the knowledge of the domain and the program under construction which is formalized in the program synthesis system. Since it is virtually impossible to re-generate this information from the synthesized program only, our approach is much more powerful and "smarter" than a certifying compiler and allows us to certify complex properties for mid-sized programs fully automatically. It also overcomes the burden of manually annotating the program; the expansion of the original 290 lines of code in our example into more than 1,700 lines of code with annotations is a clear indication that manual annotation is out of question.

This independent verification complements the notion of "correctness-by-construction" generally built into program synthesis/generation systems. This notion means that the system always produces code which correctly implements the user's specification. However, its validity depends on the correctness and consistency of the underlying synthesis engine and the domain theory. Because these are large and complex artifacts—comparable to a compiler—current technology cannot *guarantee* their correctness. Thus, a user must in reality "trust" that the synthesis system produces correct code. Here, our approach provides a means to formally and automatically (i.e., at no cost for the user) demonstrate trustworthiness of the synthesized program, even if the synthesizer itself might not be fully trustworthy.

The work described in this paper is only a first step towards this goal. In our current prototype, the safety-policy is hard-coded in the way the annotations for each individual property are generated within the synthesis schemas. We work explicit representations of safety policies (e.g., using higher-order formulations) and their use to tailor the anno-

tation generation in AUTOBAYES.

Although our approach has the potential to increase trustworthiness of the synthesis system and the code-generator, our architecture still relies on the correctness of E-SETHEO and MOPS. We are planning to implement a small and trustworthy verification condition generator and a small and verified proof checker which is able to give us the certainty that the proofs produced by E-SETHEO are indeed correct. Future work will also address issues related to proof-carrying code, in particular, a compact representation of the proofs and performing the proofs on annotated object-code.

With the emerging feasibility for the automatic generation of safety critical, non-trivial software components (e.g., for navigation/state estimation [24]), our approach to certification is able to substantially increase trustworthiness of automatically synthesized code and facilitates the use of synthesis systems in proof-carrying-code environments.

References

[1] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proc. 27th ACM Symp. Principles of Programming Languages*, pp. 243–253, ACM Press, 2000.

[2] CASC-JC theorem proving competition, 2001. `http://www.cs.miams.edu/~tptp/CASC/JC`.

[3] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. *ACM SIGPLAN Notices*, 35(5):95–107, 2000.

[4] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Engineering*, 27(2):1–25, Feb. 2001.

[5] M. S. Feather and M. Goedicke (eds.) *Proc. 16th Intl. Conf. Automated Software Engineering*. IEEE Comp. Soc. Press, 2001.

[6] B. Fischer and J. Schumann. AutoBayes: A system for generating data analysis programs from statistical models. *J. Functional Programming*, 2002. To appear. `http://ase.arc.nasa.gov/people/fischer/`.

[7] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In J. Oliveira and P. Zave (eds.), *Proc. Intl. Symp. Formal Methods Europe 2001: Formal Methods for Increasing Software Productivity*, LNCS 2021, pp. 500–517, Springer, 2001.

[8] G. C. Gannod, Y. Chen, and B. H. C. Cheng. An automated approach for supporting software reuse via reverse engineering. In D. F. Redmiles and B. Nuseibeh (eds.), *Proc. 13th Intl. Conf. Automated Software Engineering*, pp. 79–86, IEEE Comp. Soc. Press, 1998.

[9] P. Gill, W. Murray, and M. Wright. *Practical Optimization*. Academic Press, 1981.

[10] L. Hornof and T. Jim. Certifying compilation and run-time code generation. *Higher-Order and Symbolic Computation*, 12(4):337–375, 1999.

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

[12] A. Kennedy. *Programming Languages and Dimensions*. PhD thesis, University of Cambridge, Apr. 1996. Published as UCCL TR391.

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

[14] M. Lowry, T. Pressburger, and G. Rosu. Certifying domain-specific policies. In Feather and Goedicke [5], pp. 118–125.

[15] D. C. Luckham and N. Suzuki. Verification of array, record, and pointer operations in Pascal. *ACM Trans. Programming Languages and Systems*, 1(2):226–244, Oct. 1979.

[16] G. C. Necula and P. Lee. Efficient representation and validation of logical proofs. In *Proc. 13th Annual IEEE Symp. Logic in Computer Science*, pp. 93–104, IEEE Comp. Soc. Press, 1998.

[17] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C*. Cambridge Univ. Press, Cambridge, UK, 2nd. edition, 1992.

[18] M. Rittri. Dimension inference under polymorphic recursion. In *Proc. 7th Conf. Functional Programming Languages and Computer Architecture*, pp. 147–159, ACM Press, 1995.

[19] F. B. Schneider. Enforceable security policies. Computer Science Technical Report TR98-1644, Cornell University, September 1998.

[20] Z. Shao, C. League, and S. Monnier. Implementing typed intermediate language. In *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pp. 313–323, Baltimore, Maryland, Sept. 1998.

[21] M. Whalen, J. Schumann, and B. Fischer. AutoBayes/CC — combining program synthesis with automatic code certification (system description). In A. Voronkov (ed.), *Proc. 18th Intl. Conf. Automated Deduction*, LNAI 2392, pp. 290–294. Springer, 2002.

[22] 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*, LNCS 2391, pp. 431–450. Springer, 2002.

[23] M. Whalen, J. Schumann, and B. Fischer. Synthesizing certified code. RIACS Technical Report 03.02, 2002. `http://www.riacs.edu`.

[24] J. Whittle, J. Van Baalen, J. Schumann, P. Robinson, T. Pressburger, J. Penix, P. Oh, M. Lowry, and G. Brat. Amphion/NAV: Deductive synthesis of state estimation software. In Feather and Goedicke [5], pp. 395–399.

[25] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction.* The MIT Press, 1993.

[26] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proc. ACM Conf. Programming Language Design and Implementation 1998*, pp. 249–257. ACM Press, 1998. Published as SIGPLAN Notices 33(5).