Synthesizing Certified Code

Michael Whalen, Johann Schumann, and Bernd Fischer

RIACS Technical Report 02.03

03.2002

Synthesizing Certified Code[1]

Michael Whalen
Department of Computer Science and Engineering
Univ. of Minnesota, Minneapolis, MN 55455

Johann Schumann, Bernd Fischer
RIACS / NASA Ames, Moffett Field, CA 94035

RIACS Technical Report 02.03
03.2002

Code certification is a lightweight approach to demonstrate software quality on a formal level. Its basic idea is to require code producers to provide formal *proofs* that their code satisfies certain quality properties. These proofs serve as *certificates* which can be checked independently. Since code certification uses the same underlying technology as program verification, it also requires many detailed annotations (e.g., loop invariants) to make the proofs possible. However, manually adding these annotations to the code is time-consuming and error-prone.

We address this problem by combining code certification with automatic program synthesis. We propose an approach to generate simultaneously, from a high-level specification, code and *all* annotations required to certify the generated code. Here, we describe a certification extension of AutoBayes, a synthesis tool which automatically generates complex data analysis programs from compact specifications. AutoBayes contains sufficient high-level domain knowledge to generate detailed annotations. This allows us to use a general-purpose verification condition generator to produce a set of proof obligations in first-order logic. The obligations are then discharged using the automated theorem prover E-Setheo. We demonstrate our approach by certifying operator safety and memory safety for a generated iterative data classification program without manual annotation of the code.

[1] This work was supported in part by the National Aeronautics and Space Administration (NASA) under Cooperative Agreement NCC 2-1006 with the Universities Space Research Association (USRA).

Synthesizing Certified Code

Michael Whalen
Department of Computer Science and Engineering
Univ. of Minnesota
Minneapolis, MN 55455
whalen@cs.umn.edu

Johann Schumann
RIACS / NASA Ames
M/S 269-3
Moffett Field, CA 94035
schumann@email.arc.nasa.gov

Bernd Fischer
RIACS / NASA Ames
M/S 269-2
Moffett Field, CA 94035
fisch@email.arc.nasa.gov

ABSTRACT

Code certification is a lightweight approach to demonstrate software quality on a formal level. Its basic idea is to require code producers to provide formal *proofs* that their code satisfies certain quality properties. These proofs serve as *certificates* which can be checked independently. Since code certification uses the same underlying technology as program verification, it also requires many detailed annotations (e.g., loop invariants) to make the proofs possible. However, manually adding these annotations to the code is time-consuming and error-prone.

We address this problem by combining code certification with automatic program synthesis. We propose an approach to generate simultaneously, from a high-level specification, code and *all* annotations required to certify the generated code. Here, we describe a certification extension of AUTOBAYES, a synthesis tool which automatically generates complex data analysis programs from compact specifications. AUTOBAYES contains sufficient high-level domain knowledge to generate detailed annotations. This allows us to use a general-purpose verification condition generator to produce a set of proof obligations in first-order logic. The obligations are then discharged using the automated theorem prover E-SETHEO. We demonstrate our approach by certifying operator safety and memory safety for a generated iterative data classification program without manual annotation of the code.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Program Verification; I.2.2 [**Artificial Intelligence**]: Deduction and Theorem Proving; I.2.3 [**Artificial Intelligence**]: Automatic Programming

Keywords

automatic program synthesis, program verification, code certification, proof-carrying code, automated theorem proving.

1. INTRODUCTION

Code certification is a lightweight approach to demonstrate software quality on a formal level. It concentrates on certain aspects of software quality which 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 a certification authority, for example the FAA.

Code certification is an alternative to other, more established validation and verification techniques. It is more formal than code inspection and can show stronger properties. In contrast to testing, code certification demonstrates that the properties of interest hold for all possible execution paths of the program. It also complements software model checking which works on a different set of properties (typically liveness properties as for example absence of deadlocks). Moreover, model checking does not produce explicit certificates: while it can produce counter-examples if a property is violated, validity follows only indirectly from the claim of an exhaustive search through the state space which cannot be checked independently.

In essence, code certification is a more tractable version of traditional axiomatic or Hoare-style program verification. It uses the same basic technology: the program is annotated with an axiomatic specification, the annotated program is fed into a verification condition generator (VCG) which produces a series of proof obligations, the proof obligations are proven or *discharged* by an automated theorem prover (ATP). The difference, however, is in the details: the certified properties are much simpler and much more regular than full behavioral specifications. Both aspects are crucial: Since the properties are much simpler, the resulting proof obligations are much simpler as well. Consequently, discharging them is also much easier; in many cases, all proof obligations can be shown fully automatically. Since the properties are much more regular, the annotations can be derived schematically from a formulated *safety policy*. Consequently, the specification effort—which can become overwhelming in traditional verification—is also much smaller.

In an ideal world, all certification steps (i.e, annotation, verification condition generation, and proof) become part of the compilation process. A *certifying compiler* (e.g., Touchstone [5] or Cyclone [13]) thus usually comprises three components: VCG, ATP, and the proper compiler. The VCG is often specialized with respect to the safety policy support-

ed by the compiler, or, more precisely, the safety policy is hardcoded into the VCG. Hence, it can automatically insert the necessary annotations into the program.

However, code certification shares not only the underlying technology with Hoare-style program verification but also a fundamental limitation: in order to certify non-trivial programs or non-trivial properties, auxiliary annotations (e.g., loop invariants) are required. Since these annotations describe program-specific properties, the VCG cannot derive them automatically from the safety policy; instead, they must be provided manually by the software designer. This severely limits the practical usability of certification approaches like proof-carrying code (PCC) [23].

In this paper we address this problem by combining code certification with automatic program synthesis. Program synthesis (cf. [16] for an overview) is an approach to automatically generate executable code from high-level specifications. Our basic idea is to generate not only code but also *all* annotations required to certify the generated code. We will illustrate that a program synthesis system formalizes enough high-level domain knowledge from which annotations can be generated such that more complicated safety policies can be certified for more complicated programs. This domain knowledge cannot be recovered from the program by a certifying compiler as used in PCC. We will further illustrate that state-of-the-art automated theorem provers can solve the verification conditions arising from the certification of such automatically synthesized annotated code. The work reported in this paper describes an important step towards our long-term goal of extending a program synthesis system such that all generated programs can be certified completely automatically, thus relieving the users from having to annotate their code.

The remainder of the paper is organized as follows. In Section 2, we briefly describe methods and techniques underlying our approach: property verification, proof carrying code, and program synthesis. Section 3 contains a detailed architectural description of the certifying synthesizer. We specifically focus on how the safety-policy is reflected in extended Hoare-rules and how the annotations are produced and propagated during the synthesis process. We furthermore give a short description of the automated prover E-SETHEO and discuss results from processing the generated verification conditions. Section 4 covers related work and in Section 5 we conclude and sketch out future work.

2. BACKGROUND

Since this paper combines different areas, we first give some background on program verification and certification, on the notion of proof-carrying code, and on automated program synthesis, in particular the AUTOBAYES-system.

2.1 Property Verification

Traditionally, program verification has focused on showing the functional equivalence of (full) specification and implementation. However, this verification style is still too demanding, because of the involved specification and proof efforts, respectively. Therefore, more recent approaches concentrate on showing specific properties that are important for software safety. For example, model checking has been used successfully to verify liveness and safety aspects of distributed software systems [30]. We extend this property-oriented verification style in two key aspects. First, we use automated theorem provers for full first-order logic that do not require abstractions and that produce the necessary "real" proofs that can be checked independently. Second, we investigate how this approach can be extended towards a broader set of properties.

While many mechanisms and tools for verifying program properties have been published, especially for distributed systems, relatively little attention has been paid to the properties themselves. The related work in this area is usually concerned with computer security [26]; we are interested in all "useful" properties. To help guide our research, we have created an initial taxonomy of verifiable aspects of programs, as shown in Figure 1. This list is, however, only a first attempt at describing the universe of verifiable program aspects.

We first distinguish between *functional* and *property-based* verification. Functional verification is necessary to show that a program correctly implements a high-level specification. Typically, these proofs are performed by showing that a program is equivalent to, or a refinement of, some higher level specification. Property-based verification, on the other hand, ensures that the programs have desirable features (e.g. absence of runtime errors), but does not show program correctness in the traditional sense. These properties are often simpler to verify than full functional correctness; in fact, many of the properties are necessary to show functional correctness. These properties can be grouped into four categories: safety, resource-limit, liveness, and security.

Safety properties prevent the program from performing illegal or nonsensical operations, such as attempting to access memory not allocated to the program or divide a number 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. [18]). For languages with type hierarchies, it is also possible to check that all narrowing typecasts are safe and will not cause runtime errors.

Numeric safety properties assert that programs will perform arithmetic correctly. Potential errors include: (1) using partial operators, like divide or square root, with arguments outside their defined domain (e.g., 5/0), (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

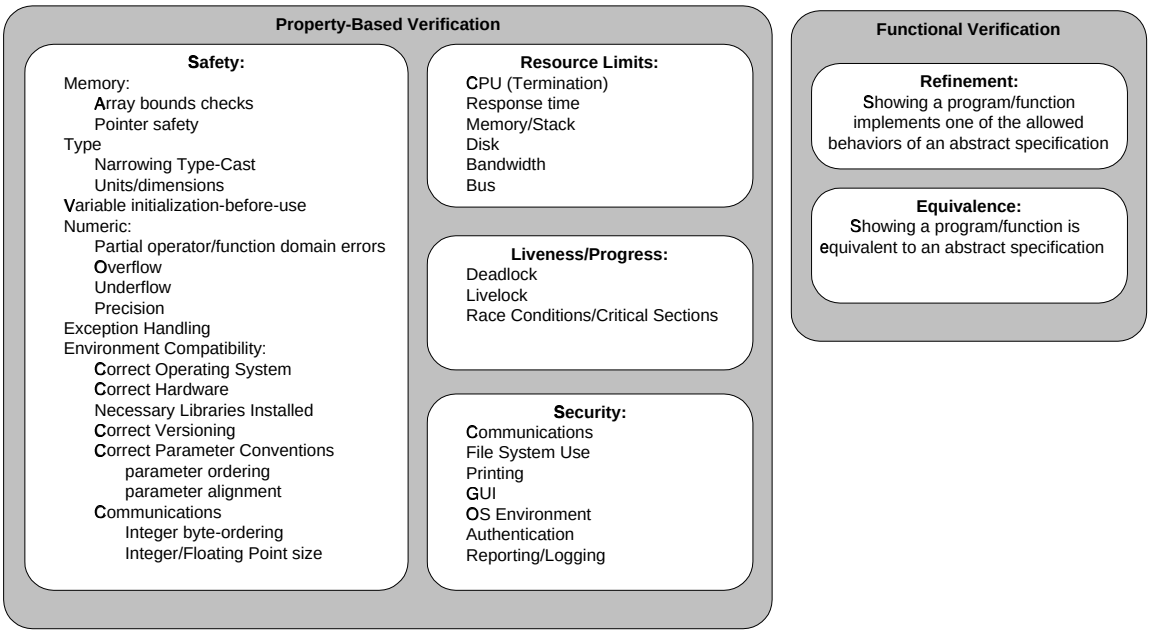

Figure 1: Property-based verification and functional verification

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. Security policies regulate access to system resources, and are often enforced by authentication procedures, which determine the identity of the program or user involved.

Clearly, there is overlap between the categories; for example, many security flaws are due to safety violations. Our list also includes many properties that are difficult or impossible to automatically verify in the general case. We plan to extend and clarify this taxonomy in future work. For this project, we are interested in properties that are amenable to automatic verification but still useful to verify. Initially, we have chosen to investigate two safety properties: array bounds checks and numeric partial operator/function domain errors.

2.2 Proof-Carrying Code

Proof-carrying code [23, 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, however, the origin of this code is unknown, or the source is not trustworthy, this poses a considerable risk: the dynamically loaded code may not be compatible with the current system status (e.g., operating system version, available resources), or the code can destroy (on purpose or not) critical data.

The concept of proof-carrying code and an accompanying system architecture (Figure 2) have been developed to address the problem of showing certain properties (i.e., a safety policy) efficiently at the time when the software is downloaded. The developer of the software annotates the program which is subsequently compiled into object-code using a certifying compiler. Such a compiler (e.g., Touchstone [5]) carries over the annotation into annotations on the object code level. A verification condition generator 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 (by a theorem prover), 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 local 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 [23] or HOL [1]). However, as mentioned earlier, all of these approaches are in practice restricted to very simple properties. More intricate properties require the producer of the program to provide elaborate annotations and to carry out complicated formal proofs manually.

2.3 Program Synthesis

Automated program synthesis aims at automatically constructing executable programs from high-level specifications. Although a variety of approaches exist [16], we will focus in

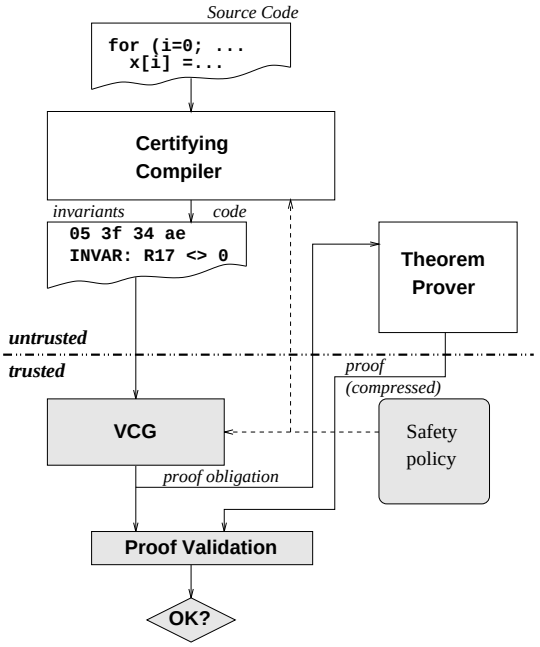

Figure 2: Typical architecture for proof carrying code. Trusted components are shaded.

this paper on a specific system, AUTOBAYES [8]. AUTOBAYES generates complex data analysis programs (currently up to 1200 lines of C++ code) from compact specifications. In the following, we will introduce its application domain by means of an example and will describe the main features of the underlying synthesis machinery.

EXAMPLE 1. *Throughout this paper, we will use the following simple but realistic classification example. Our task is to analyze spectral data measurements, e.g., from a star. Our instrument registers photons and their energy. All we know is that a photon originates from one of M different sources which emit photons at different energy levels. The energy of each photon is not sharply defined but described by a normal distribution with a certain mean value and standard deviation. However, we do not know the mean and standard deviation for each source, nor do we know their relative strength (i.e., the percentage of photons coming from each individual source). Figure 3 shows an example data set for $M = 3$ (see [3] for the physical background).*

A statistical model can be written down easily and in a compact way. For the measurements $x_0, ..., x_{N-1}$ we know that each point is normal distributed (Gaussian) around the mean value μ with a standard deviation σ for the class (individual source) c_i to which the photon belongs, i.e., $x_i \sim N(\mu_{c_i}, \sigma^2_{c_i})$ These class assignments c_i and the relative class percentages ρ are not known. All we know is that all photons belong to one of the classes, i.e., $\sum_i^M \rho_i = 1$, and that summing up the class assignments results in the desired percentages. These four formulas comprise the core of the problem specification.

An implementation for this specification, however, requires an iterative numerical algorithm[1] *which approximates the values of the desired variables μ, σ, and ρ.*

AUTOBAYES *takes a specification similar to the formulas above (a total of 19 lines including all declarations, see [8] for details) and generates executable C++ code of roughly 380 lines (including comments but not annotations) in less than a second on a SunBlade1000.* ◇

[1] In our case, an EM (expectation maximization) algorithm is generated.

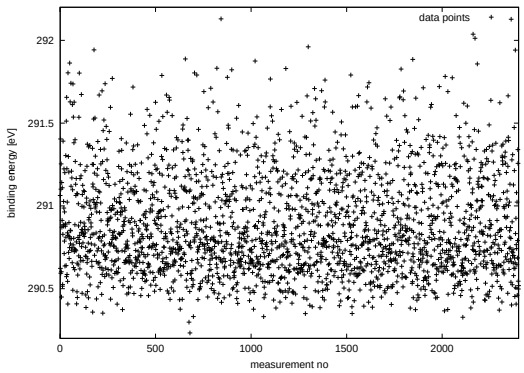

Figure 3: Example spectral data for three sources ($M = 3$). The parameters are $\mu_1 = 290.7, \sigma_1 = 0.15, \rho_1 = 0.61$, $\mu_2 = 291.13, \sigma_2 = 0.18, \rho_2 = 0.33$, and $\mu_3 = 291.55, \sigma_3 = 0.21, \rho_3 = 0.06$ (modeled after the spectrum of NH_3 molecules)

AUTOBAYES synthesizes code by exhaustive, layered 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 program calling schemas in a recursive way. 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.

Let us consider the schema which is automatically selected as the core to solve our example. This schema, presented in a Prolog notation in Figure 4 below, solves the task to estimate the desired parameters (μ, ρ, σ in our example) by maximizing their probability with respect to certain random variables (for a detailed description of the statistical background see [9]).

This information is provided as a formal input parameter to the schema in Figure 4. The output parameter *Code_fragment*↑ returns the synthesized code. After checking that this schema can be applied, the parts of the EM-algorithm are assembled. First, we generate a randomized initialization for the array `q`, then the iteration code starts. As in most numerical optimization algorithms (cf. [12, 24]), we update the local array `q` until we have reached our desired accuracy (abbreviated as `while-converging`). The code fragments for the initialization of `q` and to calculate the updates (M-Step and E-Step) are constructed by recursively calling schemas on the respective subproblems. In Figure 4,

these parts are included in ⟨...⟩. Text set in typewriter font denotes code fragments in the target language; underlined words (like max) are keywords from AUTOBAYES's specification language.

```
schema( max P(U|V) wrt V, Code_fragment↑ ) :-
      ... (* applicability constraints *)
   → Code_fragment =
     begin
      (* Initialize: *)
       ⟨guess values for c[i]⟩
      for i:=1 to N do for j:=1 to M do
        q[i,j] := 0;
      for k:=1 to M do
        q[k,c[k]] := 1;
      while-converging(V) do
       (* M-step: *)⟨ max P({q,U}|V) wrt V⟩
       (* E-step: calculate P(q|{U,V}) *)
        q[i,j] := ⟨...⟩
      end (* end while-converging *)
     end
```

Figure 4: EM-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. SYSTEM ARCHITECTURE

The architecture of our certifying synthesis system is similar to the typical proof-carrying code architecture as shown in Figure 2. However, since we are currently not dealing with proof validation aspects, we only have three major building blocks (Figure 5): the synthesis system AUTOBAYES (which replaces the certifying compiler), the verification condition generator MOPS, and the automated theorem prover E-SETHEO. All system components used in certification will be described in some more detail below.

The system's input is a statistical model which defines the data analysis task as shown above. 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 (cf. Section 3.3) to support the automatic generation of code annotations. AUTOBAYES produces Modula-2 code[2] which carries the annotations as comments. Annotations and code are then processed by the verification condition generator MOPS. 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 (basically defining all operations and functions in the proof tasks) must be added to the formulas. Finally, these extended proof obligations are fed into the automated theorem prover E-SETHEO.

For our prototype implementation, we added several small "glue" modules which convert the syntactical representation between all components. These are implemented in `lex/yacc`, `awk`, and Unix shell `sh`.

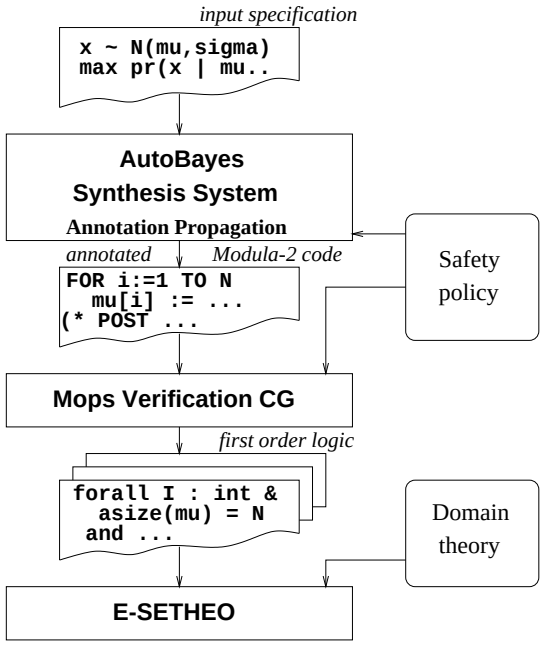

Figure 5: AutoBayes system architecture, extended for code certification

3.1 Safety Policy

The first step in certification is to define precisely what constitutes safe behavior for the programs. In our case, we must first make precise our informal notions of memory and operator safety 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. In the sections below, we formulate the safety properties and describe the mechanism for creating the verification conditions.

Hoare rules [34] form the foundation of our approach. Hoare rules 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 memory safety and operator safety constraints to the predicates in the rules to ensure that each statement is memory and operator safe. For the most part, these modifications are in the form of strengthened preconditions for each of the rules, as shown in Figure 6.

3.1.1 Array Bounds Safety Policy

[2]Since MOPS works on Modula-2, we extended AUTOBAYES to generate Modula-2 code. Usually, AUTOBAYES synthesizes C++/C programs for Octave [22] and Matlab [20].

To show array bounds safety, we first need some notion of array bounds within the assertion language, an aspect that is traditionally ignored within Hoare-style proofs. This is accomplished by defining variables within the assertion language to refer to the array bounds. Once we have these, we can test that each subscript expression within a statement is within the appropriate dimension for the array.

More concretely, we assume that given an array x of dimentsion $n+1$, all array dimensions $0 \leq k \leq n$ have the lower bound zero, and an upper bound that is represented by $ASIZE(x,k)$. The $ASIZE(x,k)$ notation is used to denote the unique variable representing that array bound.

We can then determine what it means for an expression to be array-bounds safe. Given an expression E, we say that it is array bounds safe if every array subscript expression for variable x in dimension k is between zero and $ASIZE(x,k)-1$. To check, we define a function $ArrayRefs(E)$ that returns a set of pairs of the form $(x, \langle e_0, e_1, \ldots, e_n \rangle)$. Each pair describes an array reference, where x is the array variable and $\langle e_0, e_1, \ldots, e_n \rangle$ is the sequence of subscript expressions used to access the array. Then, we can define a safe array reference $SafeRef(x, \langle e_0, e_1, \ldots, e_n \rangle)$ as:

$$SafeRef(x, \langle e_0, e_1, \ldots, e_n \rangle) = \forall a : 0..n \bullet 0 \leq e_a < ASIZE(x,a)$$

From this predicate, we can define expression safety w.r.t. the array bounds safety policy as follows:

$$SafeExpr_A(E) = \forall (x, seq) \in ArrayRefs(E) \bullet SafeRef(x, seq)$$

These two predicates state that, for each array reference, every subscript expression is within bounds.

Unfortunately, the $SafeRef$ and $SafeExpr_A$ predicates are higher-order, as they quantify over expressions in the program syntax. However, Since $ArrayRefs(E)$ 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:

$$q[k, c[k]] := 1/v; \quad (1)$$

$SafeExpr_A(E)$ yields the following safety predicate, once expanded:

$$0 \leq k \wedge k < ASIZE(q,0) \wedge 0 \leq c[k] \wedge c[k] < ASIZE(q,1) \wedge 0 \leq k \wedge k < ASIZE(c,0)$$

By checking that all array subscripts are within bounds for each array reference for each expression, we can determine array bounds safety.

3.1.2 Operator Safety Policy

To show operator safety, we only need to show that all divisors are different from zero. All other partial operators, such as square root, are implemented as standard library functions, and not as programming language language constructs. The domain constraints on these functions are enforced by standard pre- and post-conditions.

To check for zero divisors, we define a function $divisors(E)$, which returns the set of subexpressions of E used as divisors within the expression. From this function, we can define expression safety with respect to the operator safety policy as follows:

$$SafeExpr_O(E) = \forall e \in divisors(E) \bullet e \neq 0$$

So, given the statement (1), $SafeExpr_O(E)$ yields the following safety predicate, once expanded:

$$v \neq 0$$

3.1.3 Extended Hoare rules

Now that we have defined expression safety with respect to operator safety and array bounds safety, we can extend the Hoare rules with respect to these policies. First, we define $SafeExpr(E)$ as:

$$SafeExpr(E) = SafeExpr_A(E) \wedge SafeExpr_O(E)$$

Then the Hoare rules can be formulated as shown in Figure 6.

The first rule applies to array declarations. Declarations are usually ignored in Hoare-logic, but in our case a declaration rule is required to describe array bounds. As described above, we create a variable $ASIZE(x,k)$ to refer to the size of array x at dimension k. The declaration of an array is an assignment of values to these variables. The rule works by replacing instances of $ASIZE(x,k)$ in the postcondition of a statement with the variable declaration expression. For example, given an array declaration:

$$\texttt{var } c : \texttt{array}[nclasses] \texttt{ of REAL}$$

and a postcondition $ASIZE(x,0) = nclasses$, this rule would generate the following precondition:

$$nclasses = nclasses$$

which is obviously true. Through substitution, this rule provides the necessary support to reason about the size of array bounds.

It is worth noting that a seemingly more intuitive scheme, based on explicit equalities rather than substitution application, is incomplete. In this case, we could define $ASIZE(x,k)$ as a function, and the precondition of this rule to be:

$$P \wedge ASIZE(x,0) = e_0 \wedge \ldots \wedge ASIZE(x,n) = e_n$$

Unfortunately, this approach requires that $ASIZE(x,0) = e_0$, etc., are provable from the postcondition of the previous statement or the functional precondition, which is not possible.

The second rule, assignment of scalars, is the same as the standard Hoare assignment rule, except that it has a strengthened precondition that checks that the assignment expression is safe w.r.t. our safety policies.

The third 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 [19]. 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.

Array Declaration Rule:

$$\left\{\begin{array}{l} P[e_0/ASIZE(x,0),\ e_1/ASIZE(x,1),\ \ldots,\ e_n/ASIZE(x,n)]\wedge \\ \quad SafeExpr(e_0)\wedge SafeExpr(e_1)\wedge \\ \quad \vdots \\ \quad SafeExpr(e_n) \end{array}\right\} \texttt{var } x : \texttt{array}[e_0, e_1, \ldots, e_n] \texttt{ of } Y\ \{P\}$$

Scalar Assignment Rule: $\{P[e/x] \wedge SafeExpr(e)\}\ x := e\ \{P\}$

Array Assignment Rule: $\left\{\begin{array}{l} P[x\{(e_0, e_1, \ldots, e_n) \rightarrow e\}]\wedge \\ \quad SafeExpr(e)\wedge \\ \quad SafeExpr(x[e_0, e_1, \ldots, e_n]) \end{array}\right\} x[e_0, e_1, \ldots, e_n] := e\ \{P\}$

Conditional Statement Rule: $$\frac{\{P \wedge b \wedge SafeExpr(b)\}\ c\ \{Q\} \qquad (P \wedge \neg b \wedge SafeExpr(b) \implies Q)}{\{P \wedge SafeExpr(b)\}\ \texttt{if } b \texttt{ then } c\ \{Q\}}$$

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

For Loop Rule: $$\frac{\left\{\begin{array}{l} P \wedge e_0 \le x \le e_1 \\ \quad \wedge SafeExpr(e_0) \\ \quad \wedge SafeExpr(e_1) \end{array}\right\}\ C\ \left\{\begin{array}{r} P[(x+1)/x] \wedge SafeExpr(e_0) \\ \wedge SafeExpr(e_1) \end{array}\right\}}{\left\{\begin{array}{r} P[e_0/x] \wedge e_0 \le e_1 \wedge SafeExpr(e_0) \\ \wedge SafeExpr(e_1) \end{array}\right\}\ \texttt{for } x := e_0 \texttt{ to } e_1 \texttt{ do } C\ \left\{\begin{array}{r} P[(e_1+1)/x] \wedge SafeExpr(e_0) \\ \wedge SafeExpr(e_1) \end{array}\right\}}$$

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

Rule of Consequence: $$\frac{P' \implies P \quad \{P\}\ C\ \{Q\} \quad Q \implies Q'}{\{P'\}\ C\ \{Q'\}}$$

Figure 6: Hoare rules with safety policy extensions

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 (but *not* their statement bodies) are safe. The safety of the statement bodies can be shown by recursive application of the Hoare rules. These applications will generate the necessary safety predicates as a precondition to the loop or if-statement bodies. 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.

3.2 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 approach all required annotations are generated so that *any* VCG can be used. For our experiments, we used the VCG of the *Modula Proving System* MOPS [14]. MOPS is a Hoare-calculus based verification system for a large subset of the programming language Modula-2 [35], including pointers, arrays, and other data structures. The only language constructs not supported are variant record types, procedure types, and procedures as parameters. The verification of REAL-arithmetics is idealized and ignores possible rounding errors. MOPS supports the verification of arbitrary program segments and not only procedures or modules. The verification segments can be nested to break large proofs into manageable pieces.

MOPS uses a subset of VDM-SL [6] as its specification language; this is interpreted here only as syntactic sugar for classical first-order logic. All annotations are written as Modula-2 comments enclosed in `(*{...}*)`. Pre- and postconditions start with the keywords `pre` and `post`, respectively, loop invariants with a `loopinv`, and additional assertions with an `assert`.

3.3 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. The schema-guided synthesis mechanism of AUTOBAYES makes it easy to produce annotations *local* to the current statement, as the generation of annotations is tightly coupled to the individual schema. For this reason, we split the task of creating the statement annotations into two parts: creating local annotations during the run of AUTOBAYES, and propagating the annotations through the code.

3.3.1 Local Annotations

The local annotations for a statement describe the changes in variables made by the statement, without needing to describe all of the global information that may later be necessary for proofs.

EXAMPLE 2. *The code fragment which performs the initialization of the intermediate arrays (*`q` *and* `c`*) is defined in the schema described in Figure 4. The following listing shows the corresponding code fragment from the annotated schema.*

```
01 (*{ pre
02      (forall a: int &
03        (0 <= a and a < N) => 0 <= c[a] <= M) }*)
04 (*{ loopinv
05      0 <= i and i <= N - 1 and
06      0 <= j and j <= M - 1 and
07      (forall a,b : int &
08        ((0 <= a and a < i) and
09      (0 <= b and b < j)) => q[a,b] = 0.0) }*)
10 FOR i := 0 TO N - 1 DO
11   FOR j := 0 TO M - 1 DO
12      q[i,j] := 0.0;
13   END;
14 END;
15 (*{ assert
16      i = N and j = M and
17      (forall a,b : int &
18        ((0 <= a and a < N) and (0 <= b and b < M))
19         => q[a,b] = 0.0) }*)
20 (*{ loopinv
21      0 <= k and k <= N - 1 and
22      (forall a, b: int &
23        ((0 <= a and a < N) and
24         (0 <= b and b < M)
25         => 0 <= q[a,b] and q[a,b] <= 1.0) }*)
26 FOR k := 0 to N - 1 DO
27    q[k,c[k]] := 1.0;
28 END
29 (* post
30      (forall a,b : int &
31        ((0 <= a and a < N) and
32         (0 <= b and b < M))
33         => 0 <= q[a,b] and q[a,b] <=  1.0) }*)
```

◇

During synthesis (i.e., at the time when the schemas are instantiated), the annotations are produced locally for each statement. Each loop is annotated with a schematic invariant and schematic pre- and postconditions describing how it changes variables within the program. The specific form of the invariants and assertions depends on the safety policy supported by the synthesizer. For example, the precondition in lines 1–3 is required to show memory safety, more specifically, the safety of the nested array access in line 27. The fact that this precondition is actually required is part of our domain knowledge and thus encoded within the schema. Obviously, a modification or extension of the supported safety policy requires corresponding modifications or extensions of the schemas.

3.3.2 Propagation of Annotations

Unfortunately, these local annotations are in general insufficient to prove the postcondition at the end of a larger code fragment. For example, at line 27 in the code, we do not necessarily know what invariants held prior to the loop. To overcome this problem, we *propagate* any unchanged information through the annotations. Because program synthesis restricts aliasing to few, known places, the test which statements influence which annotations can be accomplished easily without full static analysis of the synthesized program.

EXAMPLE 3. *In our example, we propagate the initial condition about the vector* `c` *(lines 1–3) and add it to the loop invariant and post-assertion for the first loop (lines 15–19). Since the second loop does not change variable* `c`*, this condition is propagated forward into invariant and post-condition of the second loop.* ◇

Generally, the propagation algorithm works by creating a tree (V, E) with vertices V and edges E. The vertices V are initially labeled with the AUTOBAYES-generated local annotations. The edges E describe the locations of the annotations relative to one another in the code according to a lexicographic ordering which obeys the nesting of the language constructs. Each vertex in the tree may have many children in two categories: one *sibling* vertex and zero or more *child* vertices, corresponding to the lexical placement of the annotations in the code. The edges are labelled by the set of variables that have been assigned between the annotations.

EXAMPLE 4. *Figure 8 gives an example annotation graph for a code fragment. In this fragment, edges* (A_0, A_6) *and* (A_1, A_5) *describe sibling relationships and* (A_0, A_1), (A_1, A_2), (A_1, A_3) *and* (A_3, A_4) *describe parent/child relationships (dashed arrows). The code-fragment for this example does not refer to any variables. Hence, the edges are not labeled.* ◇

```
procedure propagate(root : Vertex,
                    inherited : predicate set)
begin
  annotation(root) := annotation(root) ∪ inherited;
  forall c in children(root)
    c_inherits := {};
    vars := vars_assigned(edge(root,c));
    forall a in annotations(root)
      if variables(a) ∩ vars = {} then
        c_inherits := c_inherits ∪ a;
    end if
    end forall
  propagate(c,c_inherits)
  end forall
end
```

Figure 7: The annotation propagation algorithm

The algorithm, shown in Figure 7, starts from the top of the tree and performs a recursive depth-first traversal. The parameters to the algorithm are the current root node of the tree (*root*) and the set of inherited formulas (*inherited*). The algorithm first updates the annotations associated with the root node, *annotation*(*root*), to include the parent-node formulas. Then, for each child, it creates a set of inherited predicates (*c_inherits*) and calls itself recursively. The set *c_inherits* is a set of all predicates from *annotation*(*root*) that do not contain variables modified by the intervening code. This information is extracted from the labeling of the edge between the root node and node *c*.

3.4 The Automated Prover

For our experiments we used the automated theorem prover E-SETHEO, version csp01 [4]. E-SETHEO is a compositional theorem prover for formulas in first order logic, combining the systems E [27] and SETHEO [21, 17]. The 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. These different schedules have been computed from experimental data using machine learning

(A)

```
(*{ A0 }*)
while B0 do
    ...
    (*{ A1 }*)
    while B1 do
        if B2 then
            ...
            (*{ A2 }*)
        end
        ...
        (*{ A3 }*)
        if B3 then
            ...
            (*{ A4 }*)
        end
    end
    (*{ A5 }*)
end
(*{ A6 }*)
```

(B)

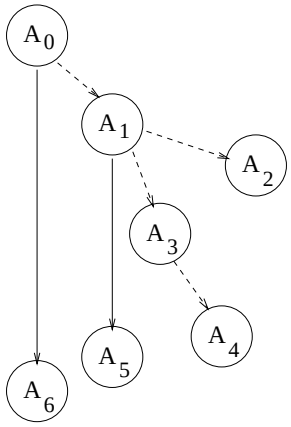

Figure 8: (A) A fragment of annotated code. (B) The annotation graph for the code.

techniques [29]. Because all of the subsystems work on formulas in clausal normal form (CNF), the first order formula is first converted into CNF using the module Flotter [32]. E-SETHEO is one of the most powerful ATP systems available as has been shown in recent international theorem proving competitions [4].

Out of the 69 proof tasks of our example, E-SETHEO could solve 65 automatically with a run-time limit of 60 seconds on a 1000 Mhz. SunBlade workstation. Most of the tasks could be solved in about one second, but several tasks took up to 40 seconds. The average elapsed proof time was 6.3 seconds. The remaining four proof tasks required some manual preprocessing and special strategies before they could be proven automatically. Due to the flexible architecture of E-SETHEO, these steps can be incorporated as new strategies. So, in the near future, we expect to be able to handle all proof tasks automatically.

4. RELATED WORK

We are not aware of any other work to automatically extract knowledge about the program under construction from the synthesis process, whether for certification or for other purposes. However, there is a large number of different approaches which share either techniques or goals with our work.

The approach most closely related to ours is proof-carrying code which has already been discussed in Section 2.2. However, due to its focus on mobile code, PCC covers many aspects we are (currently) not 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 [28]) and is thus complementary to our approach. Certifying compilers as Touchstone [5] or Cyclone [13] could consequently be used to show that the safety policy established on the source code level is not compromised by the compilation step.

Lowry et al. [18] 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 [25, 15] and memory safety [36]. 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.

Many reverse engineering approaches try to recover formal specifications from code. Gannod and Cheng [11] 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. [7] 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 [10] 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 or refinement between specifications and programs. This is true especially for integrated development/proof environments as for example the KIV system [31]. Moreover, program verification systems usually offer no support to find and to formalize the functional specifications and auxiliary annotations which was the original motivation for our approach.

It is sometimes possible to encode aspects of the safety policy into the logic used in a program verification system. For example, if VDM is not interpreted as mere syntactic sugar for classical logic but as notation for the three-valued logic of partial functions (LPF) [2], a partial correctness proof in MOPS already gives operator safety. However, it is not clear what other safety policies can be encoded into

suitable logics.

5. CONCLUSIONS

In this paper, we have described a novel combination of automated program synthesis and automated program verification. Our basic idea is to generate the program together with detailed formal annotations which are required for a fully automatic correctness proof. This approach is facilitated by the knowledge of the domain and the program under construction which are 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.

We have demonstrated the feasibility of our approach by certifying operator safety and memory safety for an automatically generated iterative data classification program. The synthesized program consists of roughly 380 lines of code, 90 of which are auto-generated comments to explain the code. With all annotations (including annotation propagation), it grows to 2,116 lines of code—a clear indication than manual annotation is out of question. The annotated program induces 69 proof tasks in first-order logic. After some minor preprocessing steps, all these tasks can be solved automatically in relatively short time, using the theorem prover E-SETHEO.

Our long-term goal is to extend AUTOBAYES such that all generated programs can be certified completely automatically. We are confident that our approach can also be extended to other program synthesis systems, because they generally encode enough abstract knowledge about the domain and the program under construction. We see a number of benefits from this combination of program synthesis and program verification. For the user of such a certifying synthesis system, the major benefit is obviously the additional verification of (important aspects of) the synthesized code; moreover, it comes at no cost, and it can be double-checked independently.

This independent verification complements the notion of "correctness-by-construction" generally built into program synthesis 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. However, for safety/security sensitive code or mission critical code, e.g., for navigation/state estimation [33], this is not acceptable. Here, our approach provides a tool and methodology to demonstrate important properties of the code in an automatic and independently re-checkable way.

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 are generated within the synthesis schemas. We work on ways to explicitly represent safety policies (e.g., using higher-order formulations) and use this to tailor the annotation generation in AUTOBAYES. Our architecture also relies on the correctness of E-SETHEO. We are planning to extend our system to incorporate a small and verified proof checker which is able to give us the certainty that the proofs produced by E-SETHEO are indeed correct. Furthermore, we plan to implement a small and trustworthy verification condition generator. Future work will also be concerned with addressing proof-carrying code related issues, in particular, a compact representation of the proofs and performing the proofs on annotated object-code.

Acknowledgements. Most of this work was done during M. Whalen's visit at NASA/Ames, which was supported by a RIACS SSRP grant.

6. REFERENCES

[1] A. W. Appel and A. Felty. A semantic model of types and machine instuctions for proof-carrying code. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-00)*, pages 243–253. ACM Press, 2000.

[2] H. Barringer, J. H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21(3):251–269, Oct. 1984.

[3] J. Berkowitz. *Photoabsorption, photoionization, and photoelectron spectroscopy.* Academic Press, 1979.

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

[5] 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.

[6] J. Dawes. *The VDM-SL Reference Guide.* Pitman, London, 1991.

[7] 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.

[8] B. Fischer and J. Schumann. AutoBayes: A system for generating data analysis programs from statistical models, 2001. Submitted for publication. Preprint available at `http://ase.arc.nasa.gov/people/fischer/`.

[9] B. Fischer, J. M. P. Schumann, and T. Pressburger. Generating data analysis programs from statistical models (position paper). In W. Taha, editor, *Proc. Intl. Workshop Semantics Applications, and Implementation of Program Generation*, volume 1924 of *Lect. Notes Comp. Sci.*, pages 212–229, Montreal, Canada, Sept. 2000. Springer.

[10] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In J. Oliveira and P. Zave, editors, *Proc. Intl. Symp. Formal Methods Europe 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *Lect. Notes Comp. Sci.*, pages 500–517, Berlin, Mar. 12–16 1997. Springer.

[11] 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, editors, *Proc. 13th Intl. Conf. Automated Software Engineering*, pages 79–86, Honolulu, Hawaii, Oct. 1998. IEEE Comp. Soc. Press.

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

[13] L. Hornof and T. Jim. Certifying compilation and

run-time code generation. *Higher-Order and Symbolic Computation*, 12(4):337–375, 1999.

[14] 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*, pages 163–167, Reisensburg, July 2000.

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

[16] C. Kreitz. Program synthesis. In W. Bibel and P. H. Schmitt, editors, *Automated Deduction - A Basis for Applications*, pages 105–134. Kluwer, Dordrecht, 1998.

[17] R. Letz, J. Schumann, S. Bayerl, and W. Bibel. SETHEO: A High-Performance Theorem Prover. *Journal of Automated Reasoning*, 8(2):183–212, 1992.

[18] M. Lowry, T. Pressburger, and G. Rosu. Certifying domain-speficic policies. In M. S. Feather and M. Goedicke, editors, *Proc. 16th Intl. Conf. Automated Software Engineering*, pages 118–125, San Diego, CA, Nov. 26–29 2001. IEEE Comp. Soc. Press.

[19] 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.

[20] C. B. Moler, J. N. Little, and S. Bangert. *PC-Matlab Users Guide*. Cochituate Place, 24 Prime Park Way, Natick, MA, USA, 1987.

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

[22] M. Murphy. Octave: A free, high-level language for mathematics. *Linux Journal*, 39, July 1997.

[23] G. C. Necula and P. Lee. Efficient representation and validation of logical proofs. In *Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS'98)*, pages 93–104. IEEE Computer Society Press, 1998.

[24] 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.

[25] M. Rittri. Dimension inference under polymorphic recursion. In *Proc. 7th Conf. Functional Programming Languages and Computer Architecture*, pages 147–159, La Jolla, CA, June 25–28 1995. ACM Press.

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

[27] S. Schulz. System abstract: E 0.3. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *Lecture Notes in Artifical Intelligence*, pages 297–301. Springer Verlag, 1999.

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

[29] G. Stenz and A. Wolf. E-SETHEO: Design configuration and use of a parallel theorem prover. In N. Foo, editor, *"Proceedings of AI'99, the 12th Australian Joint Conference on Artificial Intelligence"*, number 1747 in Lecture Notes in Artifical Intelligence, pages 231–243. Springer Verlag, 1999.

[30] W. Visser, K. Havelund, G. Brat, and S.-J. Park. Model checking programs. In *Proc. of the 15th IEEE International Conference on Automated Software Engineering*, Grenoble, France, September 2000.

[31] W. Reif. The KIV Approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages and Tools for the Construction of Correct Software*, volume 1009 of *Lect. Notes Comp. Sci.*, pages 339–370. Springer Verlag, 1995.

[32] C. Weidenbach, B. Gaede, and G. Rock. SPASS & FLOTTER version 0.42. In M. A. McRobbie and J. K. Slaney, editors, *Proceedings of the Thirteenth International Conference on Automated Deduction (CADE-96)*, volume 1104 of *Lecture Notes in Artifical Intelligence*, pages 141–145. Springer Verlag, 1996.

[33] 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 (short paper). In *Procedings of the 16th Automated Software Engineering Conference 2001 (ASE 2001)*. IEEE, 2001. to appear.

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

[35] N. Wirth. *Programming in Modula-2*. Springer, Berlin, 4th edition, 1988.

[36] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proc. ACM Conf. on Programming Language Design and Implementation 1998*, pages 249–257, Montreal, Canada, June 17–19 1998. ACM Press. Published as SIGPLAN Notices 33(5).