# A Generic Annotation Inference Algorithm for the Safety Certification of Automatically Generated Code

Ewen Denney
RIACS / NASA Ames
m/s 269-2, Moffett Field, CA 94035, USA
edenney@email.arc.nasa.gov

Bernd Fischer
School of Electronics and Computer Science
University of Southampton, England
B.Fischer@ecs.soton.ac.uk

## Abstract

Code generators for realistic application domains are not directly verifiable in practice. In the certifiable code generation approach the generator is extended to generate logical annotations (i.e., pre- and postconditions and loop invariants) along with the programs, allowing fully automated program proofs of different safety properties. However, this requires access to the generator sources, and remains difficult to implement and maintain because the annotations are cross-cutting concerns, both on the object-level (i.e., in the generated code) and on the meta-level (i.e., in the generator).

Here we describe a new generic post-generation annotation inference algorithm that circumvents these problems. We exploit the fact that the output of a code generator is highly idiomatic, so that patterns can be used to describe all code constructs that require annotations. The patterns are specific to the idioms of the targeted code generator and to the safety property to be shown, but the algorithm itself remains generic. It is based on a pattern matcher used to identify instances of the idioms and build a property-specific abstracted control flow graph, and a graph traversal that follows the paths from the use nodes backwards to all corresponding definitions, annotating the statements along these paths. This core is instantiated for two generators and successfully applied to automatically certify initialization safety for a range of generated programs.

***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

***General Terms*** Algorithms, Verification

***Keywords*** automated code generation, program verification, software certification, Hoare calculus, logical annotations, automated theorem proving

## 1. Introduction

Automated code generation is an enabling technology for model-based software development and has significant potential to improve the entire software development process. It promises many benefits, including higher productivity, reduced turn-around times, increased portability, and elimination of manual coding errors. However, the key to realizing these benefits is of course generator correctness—nothing is gained from replacing manual coding errors with automatic coding errors.

Since the direct verification of generators is still unfeasible with existing verification techniques, several alternative approaches based on "correct-by-construction" techniques like deductive synthesis [24] or refinement [23] have been explored. However, these remain difficult to implement and to scale up, and have not found widespread application. Currently, generators are thus validated primarily by testing [25], but this quickly becomes excessive and cannot guarantee the same level of assurance.

Our work follows an alternative approach that is based on the principle that the correctness of the generator is irrelevant if instead the correctness of the generated programs is shown individually. In particular, we follow the same pragmatic approach as proof carrying code (PCC) [20] and focus on the Hoare-style certification of specific safety properties rather than showing full correctness of the generated programs. This simplifies the task but it still leaves us with the problem of constructing the appropriate logical annotations (i.e., pre- and postconditions and loop invariants), due to their central role in Hoare-style techniques.

In previous work [6, 7, 10, 27], we developed and evaluated the *certifiable code generation* approach, in which the generator itself is extended in such a way that it produces the necessary annotations together with the code. This is achieved by embedding annotation templates into the code templates, which are then instantiated and refined in parallel by the generator. We have successfully used this approach to certify a variety of safety properties for code generated by the AUTOBAYES [13] and AUTOFILTER [28] systems. However, it has two major disadvantages. First, it requires access to the existing sources: the developers need to modify the code generator in order to integrate the annotation generation. Unfortunately, sources are often not accessible, in particular for commercial generators. Second, it is difficult to implement and to maintain: for each safety property, the developers first need to analyze the generated code in order to identify the location and structure of the required annotations, then identify the templates that produce the respective code fragments, and finally formulate and integrate appropriate annotation templates. This is compounded by the fact that annotations are cross-cutting concerns, both on the object-level (i.e., the generated program) and the meta-level (i.e., the generator). In our case, extensions and modifications to the code generators have over time thus led to a situation of "entropic decay" where the generated annotations have not kept pace with the generated code, and (safe) programs fail to be proven safe.

Here we describe an alternative approach that uses a generic post-generation annotation inference algorithm to circumvent these



*GPCE'06* October 22–26, 2006, Portland, Oregon, USA.

problems. We exploit both the highly idiomatic structure of automatically generated code and the restriction to specific safety properties. Since generated code only constitutes a limited subset of all possible programs, the new "eureka" insights that are required in general program verification remain rare in our case. Since safety properties are simpler than full functional correctness, the required annotations are also simpler and more regular. We can thus use code patterns to describe all code constructs that require annotations and templates to describe the annotations that are required at the pattern locations. We can then use techniques similar to aspect-oriented programming to add the annotations to the generated code: the patterns correspond to (static) point-cut descriptors, while the introduced annotations correspond to advice.

Similar to the certifiable code generation approach, we still split the problem of certifying code into two phases: an untrusted annotation construction phase, and a simpler but trusted verification phase where the standard machinery of a verification condition generator (VCG) and automated theorem prover (ATP) is used to prove that the code satisfies the safety property. However, our new algorithm concentrates annotation generation in one location and, even more importantly, leaves the generator unchanged because it can run completely separately from the generator.

The main contribution of this paper is the development of a generic approach to extending code generators with a safety certification capability. At the core of the algorithm are a pattern matcher that is used to identify instances of the idioms and to build build property-specific abstracted control flow graphs, and a graph traversal that follows the paths from the use nodes backwards to all corresponding definitions and annotates the statements along these paths. The underlying annotation inference algorithm has been applied to certify initialization safety for code generated by the AUTOBAYES and AUTOFILTER systems. The focus in this paper is on the inference algorithm and the core components, rather than the subsequent generation and proof of verification conditions. We use initialization safety as example property to illustrate the algorithm, but the algorithm itself is generic with respect to the safety property. In the next section, we briefly provide some background; for more details we refer to [6, 7, 13]. We then introduce annotation inference informally by a worked example in Section 3 before we explain the technical details of the algorithm in Section 4. In Section 5, we summarize the experiences and experimental results with applying our algorithm to code generated by AUTOBAYES and AUTOFILTER. The final two sections discuss related and future work.

## 2. Background

**Idiomatic Code** Automated code generators derive lower-level code from higher-level, declarative specifications. Approaches range from deductive synthesis [24] to template meta-programming [4] but for our purposes neither the specific approach nor the specification language matter, and we build on a template-based approach [5]. What *does* matter, however, is the fact that an automatic code generator usually generates highly *idiomatic code*. Intuitively, idiomatic code exhibits some regular structure beyond the syntax of the programming language and uses similar constructions for similar problems. Manually written code already tends to be idiomatic, but the idioms used vary with the programmer. Automated generators eliminate this variability because they derive code by combining a finite number of building blocks—in our case, templates. For example, AUTOBAYES and AUTOFILTER only use three templates to initialize a matrix, resulting in either straight-line code or one of two doubly-nested loop versions (Figure 1).

The idioms are essential to our approach because they (rather than the templates) determine the interface between the code generator and the inference algorithm. For each generator and safety property, our approach thus requires a customization step in which the relevant idioms are identified and formalized as patterns. Note that neither missed idioms nor wrong patterns can compromise the assurance given by the safety proofs because the inferred annotations remain untrusted. They can, however, compromise the "completeness" of the approach in the sense that safe programs can fail to be proven safe, and in our experience, a few iterations can be required to identify all patterns. Note also that the idioms can be recognized from a given code base alone, even without knowing the templates that produced the code. This gives us two additional benefits. First, it allows us to apply our technique to black-box generators as well. Second, it also allows us to handle optimizations: as long as the resulting code can be described by patterns neither the specific optimizations nor their order matter.

```
A[1,1]:= a_{1,1};          for i:= 1 to n do          for i:= 1 to n do
...                          for j:= 1 to m do          for j:= 1 to m do
A[1,m]:= a_{1,m};              B[i,j]:= b;                if i=j then
A[2,1]:= a_{2,1};                                           C[i,j]:= c
...                                                       else
A[n,m]:= a_{n,m};                                           C[i,j]:= c';
```

**Figure 1.** Idiomatic matrix initializations in AUTOBAYES and AUTOFILTER

**Safety Certification** The purpose of safety certification is to demonstrate that a program does not violate certain conditions during its execution. A *safety property* is an exact characterization of these conditions based on the operational semantics of the language. A *safety policy* is a set of Hoare rules designed to show that safe programs satisfy the safety property of interest. Figure 2 shows the rules of the initialization safety policy as an example. The rules are formalized using the usual Hoare triples $P\ \{c\}\ Q$, i.e., if the condition $P$ holds before and the command $c$ terminates, then $Q$ holds afterwards. For example, the *assert* rule says that given an arbitrary incoming postcondition $Q$, we must first prove that the asserted postcondition $Q'$ implies this. We then compute the *weakest precondition* (WPC) of $Q'$ for $c$ and show that the asserted precondition $P'$ implies this. The asserted precondition is then passed on as the WPC of the annotated statement. See [19] for more information about Hoare-style program proofs.

For each notion of safety the appropriate safety property and corresponding policy must be formulated. This is usually straightforward; in particular, the safety policy can be constructed systematically by instantiating a generic rule set that is derived from the standard rules of the Hoare calculus [6]. The basic idea is to extend the standard environment of program variables with a "shadow" environment of safety variables which record safety information related to the corresponding program variables. The rules are then responsible for maintaining this environment and producing the appropriate verification conditions (VCs). This is done using a family of *safety substitutions* that are added to the normal substitutions, and a family of *safety conditions* that are added to the calculated WPCs. Safety certification then starts with the outermost (i.e., at the end of the program) postcondition *true* and computes the weakest safety precondition (WSPC), i.e., the WPC together with all applied safety conditions and safety substitutions. If the program is safe then its WSPC will be provable without any assumptions.[1]

In this paper, we focus on initialization safety, which we use as our running example here but a range of other safety properties, including absence of out-of-bounds array accesses and nil-pointer

[1] As usual for the Hoare approach, the calculus is only relatively complete, so technically the derived WSPC will only be provable in a sufficiently strong logic.

(*assign*) $$\frac{}{Q[e/x, \text{INIT}/x_{\text{init}}] \wedge \mathit{safe}_{\text{init}}(e)\ \{x := e\}\ Q}$$

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

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

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

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

(*comp*) $$\frac{P\ \{c_1\}\ R \quad R\ \{c_2\}\ Q}{P\ \{c_1\ ;\ c_2\}\ Q}$$ (*skip*) $$\frac{}{Q\ \{\textbf{skip}\}\ Q}$$ (*assert*) $$\frac{P' \Rightarrow P \quad P\ \{c\}\ Q' \quad Q' \Rightarrow Q}{P'\ \{\textbf{pre}\ P'\ c\ \textbf{post}\ Q'\}\ Q}$$

**Figure 2.** Proof rules for initialization safety

dereferences, have already been formalized [6, 20] and can in principle be used with our algorithm. Initialization safety ensures that each variable or individual array element has been explicitly assigned a value before it is used. The safety environment consists of shadow variables $x_{\text{init}}$ that contain the value INIT after the variable $x$ has been assigned a value. Arrays are represented by shadow arrays to capture the status of the individual elements. The rules can be read backwards to compute the WSPCs. For example, the *for*-rule says that for an arbitrary postcondition, $Q$, if $c$ has WSPC $P$ for the postcondition $I[i+1/i]$, and if the two intermediate VCs are true, then the WSPC of the loop is as shown. Only statements assigning a value to a location affect the value of a shadow variable (i.e., the *assign*-, *update*-, and *for*-rules). However, all rules also produce the appropriate safety conditions $\mathit{safe}_{\text{init}}(e)$ for all immediate subexpressions $e$ of the statements. Since the safety property defines an expression to be safe if all corresponding shadow variables have the value INIT, $\mathit{safe}_{\text{init}}(x[i])$ for example simply translates to $i_{\text{init}} = \text{INIT} \wedge (x_{\text{init}}[i]) = \text{INIT}$.

**VC Processing and Annotations** As usual in Hoare-style verification, the VCG traverses the annotated code and applies the rules of the calculus to produce VCs. These are then simplified, completed by an axiomatization of the background theory and passed to an off-the-shelf ATP. If all VCs are proven, the program is safe with respect to the safety property. Note that the ATP has no access to the program internals; hence, all pertinent information must be taken from the annotations, which become part of the VCs. For full functional verification, annotations are thus usually very detailed and, consequently, annotation inference remains intractable for this case. For safety certification, on the other hand, the Hoare rules of the safety policy already have more internal structure and the safety conditions are regular and relatively small, so that the required annotations are a lot simpler. For example, initialization safety just requires that the logical annotations entail at each use of a variable $x$ that the corresponding shadow variable $x_{\text{init}}$ has the value INIT. In addition, the targeted safety property and policy are known at annotation inference time, which eliminates the need for any logical reasoning in the style of the early inference approaches [26].

**System Architecture** Figure 3 shows the overall system architecture of our certification approach. At its core is the original (unmodified) code generator which is complemented by the annotation inference subsystem, including the pattern library and the annotation templates, as well as the standard machinery for Hoare-style techniques, i.e., VCG, simplifier, ATP, proof checker, and domain theory. These components and their interactions are described in the rest of this paper and in more detail in [6, 10, 27]. As in the PCC approach, the architecture distinguishes between trusted and untrusted components, shown in Figure 3 in red (dark grey) and blue (light grey), respectively. *Trusted* components *must be correct* because any errors in them can compromise the assurance provided by the overall system. *Untrusted* components, on the other hand, are not crucial to the assurance because their results are double-checked by at least one trusted component. In particular, the assurance provided by our approach does not depend on the correctness of the two largest (and most complicated) components: the original code generator, and the ATP; instead, we need only trust the safety policy, the VCG, the domain theory, and the proof checker. Moreover, the annotation inference subsystem (including the pattern library and annotation templates) also remain untrusted since the resulting annotations simply serve as "hints" for the subsequent analysis steps.

## 3. A Worked Example

Before we describe the details of the inference algorithm, we illustrate it by means of a worked example. Figure 4(a) shows a simple program that initializes two vectors `A` and `B` of size `N` with given but irrelevant values $a_i$ and $b$ (see lines 2.1–2.$n$ and 3.1–3.2, respectively) and then computes and returns the sums `s` and `t` of their respective elements as well as their dot-product `d`. It is derived from and representative of the code generated by AUTOFILTER; in particular it shows the same overall structure—a series of variable definitions followed by a loop with variable uses. AUTOFILTER's target language is a simple imperative language with basic control constructs (i.e., **if** and **for**) and numeric scalars and arrays as the only datatypes. However, the language also supports domain-specific operations on entire vectors and matrices like matrix multiplication or assignment, although these are not used in the example shown in Figure 4.

The aim of the inference algorithm is to "get information from definitions to the uses", i.e., to annotate the program in such a way that the VCG will have the necessary information to show the program safe with respect to the given property as it works its way back through the program. In the example therefore we need—amongst others—an invariant for the loop at line 5.1 that ensures that the shadow variables corresponding to the scalar variables `s`, `t`, and `d` and to the arrays `A` and `B` have the value INIT.

Since the safety-relevant information is represented by the shadow variables, the inference algorithm first scans the code for the relevant corresponding program variables. For each relevant vari-

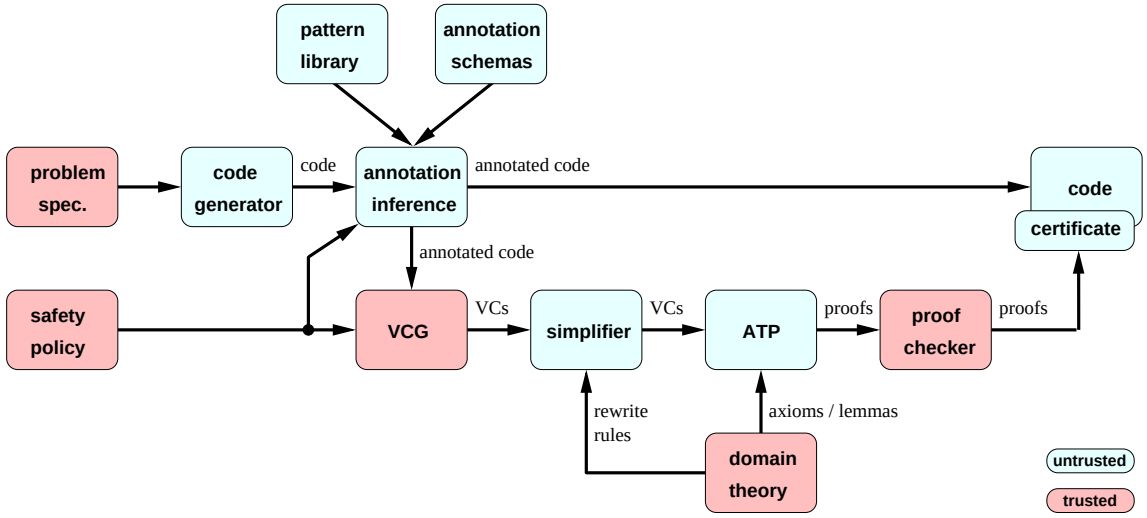


**Figure 3.** System architecture

| | (a) | (b) | (c) | (d) | (e) |
|---|---|---|---|---|---|
| 1.1 | **const** N := $n$; | **const** N := $n$; | **const** N := $n$; | *block*(A); | **const** N := $n$; |
| 1.2 | **var** i,s,t,d; | **var** i,s,t,d; | **var** i,s,t,d; | | **var** i,s,t,d; |
| 1.3 | **var** A[1:N],B[1:N]; | **var** A[1:N],B[1:N]; | **var** A[1:N],B[1:N]; | | **var** A[1:N],B[1:N]; |
| 2.1 | A[1] := $a_1$; | A[1] := $a_1$; | A[1] := $a_1$; | *def*(A[1:$N$]); | A[1] := $a_1$; |
| | ... | ... | ... | | ... |
| 2.$n$ | A[$n$] := $a_n$; | A[$n$] := $a_n$; | A[$n$] := $a_n$; | | A[$n$] := $a_n$; **post** $\forall j \in \{1:n\} \cdot A_{\text{init}}[j] = \text{INIT}$ |
| 3.1 | **for** i := 1 **to** N **do** | *def*(B[1:N]); | **for** i := 1 **to** N **inv** $\forall j \in \{1:i-1\} \cdot B_{\text{init}}[j] = \text{INIT}$ **do** | *barrier*(A); | **for** i := 1 **to** N **inv** $\forall j \in \{1:n\} \cdot A_{\text{init}}[j] = \text{INIT}$ $\wedge\ \forall j \in \{1:i-1\} \cdot B_{\text{init}}[j] = \text{INIT}$ **do** |
| 3.2 | B[i] := $b$; | | B[i] := $b$; **post** $\forall j \in \{1:N\} \cdot B_{\text{init}}[j] = \text{INIT}$ | | B[i] := $b$; **post** $\forall j \in \{1:n\} \cdot A_{\text{init}}[j] = \text{INIT}$ $\wedge\ \forall j \in \{1:N\} \cdot B_{\text{init}}[j] = \text{INIT}$ |
| 4.1 | s:=0; | s:=0; | s:=0; | *block*(A); | s:=0; |
| 4.2 | t:=0; | t:=0; | t:=0; | | t:=0; |
| 4.3 | d:=0; | d:=0; | d:=0; | | d:=0; |
| 5.1 | **for** i := 1 **to** N **do** | **for** i := 1 **to** N **do** | **for** i := 1 **to** N **inv** $\forall j \in \{1:N\} \cdot B_{\text{init}}[j] = \text{INIT}$ **do** | **for** i := 1 **to** N **do** | **for** i := 1 **to** N **inv** $\forall j \in \{1:n\} \cdot A_{\text{init}}[j] = \text{INIT}$ $\wedge\ \forall j \in \{1:N\} \cdot B_{\text{init}}[j] = \text{INIT}$ $\wedge\ s_{\text{init}} = t_{\text{init}} = d_{\text{init}} = \text{INIT}$ **do** |
| 5.2 | s:=s+A[i]; | s:=s+A[i]; | s:=s+A[i]; | *use*(A); | s:=s+A[i]; |
| 5.3 | t:=t+B[i]; | *use*(B); | t:=t+B[i]; | *block*(A); | t:=t+B[i]; |
| 5.4 | d:=d+A[i]*B[i]; | *use*(B); | d:=d+A[i]*B[i]; | *use*(A); | d:=d+A[i]*B[i]; **post** $s_{\text{init}} = t_{\text{init}} = d_{\text{init}} = \text{INIT}$ |
| 6 | **return** s,t,d; | **return** s,t,d; | **return** s,t,d; | *block*(A); | **return** s,t,d; |

**Figure 4.** (a) Original program (b) Abstraction for B (c) Annotations for B (d) Abstraction for A (using *block*- and *barrier*-patterns) (e) Fully annotated program.

able, the algorithm then builds an abstracted control flow graph (CFG) where irrelevant parts of the program are collapsed into single nodes and follows all paths backwards from the variable's use nodes until it encounters either a cycle or a definition node for the variable. Paths that do not end in a definition are discarded and the remaining paths are traversed node by node. First the definitions themselves are annotated, and then annotations are added to all intermediate nodes that otherwise constitute barriers to the information flow.

For initialization safety all variables that are used on the right-hand side of assignments (more precisely, in *rvar*-positions) are relevant, but for this example we will restrict our attention to the two array variables A and B, starting with B which is used in lines 5.3 and 5.4. Both uses are abstracted into *use*(B), in Figure 4(b). The only assignment to B is in line 3.2; however, this is not the entire definition—the algorithm needs to identify the **for**-loop (lines 3.1-3.2) as the definition for the entire array B and abstract it into the definition node *def*(B[1:N]). The path search then starts at line 5.4 and goes straight back up to the **for**-loop at line 5.1, where it splits. One branch comes in from the bottom of the loop-body but this immediately leads to a cycle and is therefore discarded. The other branch continues through lines 4.1–4.3 and terminates at the

definition node at line 3.1. Since all branches have been exhausted, there is only one path along which annotations need to be added. The annotation process starts with the use and proceeds towards the definition terminating the path. The form of all annotations is fully determined by the known syntactic structure of the definition and by the safety property. Since the definition is a (singly-nested) loop, in this case, it needs a loop invariant as well as a postcondition. Since the safety property is initialization safety, both invariant and postcondition need to formalize that the shadow variable $B_{\text{init}}$ corresponding to the current array variable B records the value INIT for the already initialized entries. Note that the different upper bounds for the quantifiers can both be constructed from the loop. The postcondition is then pulled along the remaining path, i.e., added to all nodes that require it. Every node needs to be inspected, but in this case only the **for**-loop at line 5.1 requires an invariant. Figure 4(c) shows the partially annotated program that results from this pass.

The next pass adds the annotations for A (Figure 4(d)). As before, its two uses in lines 5.2 and 5.4 are abstracted. A is initialized using a different idiom—a sequence of assignments, lines 2.1–2.$n$—which is again collapsed into a *def*-node; here, the initialized range is taken from the first and last assignment, respectively. The program is collapsed further by the introduction of *barrier*- and *block*-nodes. These represent areas that do not need to be explored because they cannot contain relevant definitions, thus in general substantially reducing the number of paths. Both are also described by property-specific patterns. However, *barrier*-nodes must be re-expanded during the path traversal phase because they require annotations (line 3.1) while *block*-nodes remain opaque. Except for this special handling, the algorithm proceeds as before, and Figure 4(e) shows the resulting fully annotated program.

## 4. Inference Algorithm

The example in the previous section shows that the set of idiomatic coding patterns which are used is the key knowledge that drives the annotation construction. Finding instances of these patterns in the code is not a general program understanding problem, however: we are not concerned with identifying general-purpose coding patterns and cliches [22] but only the relevant definitions and uses. These are specific to the given safety property, but the algorithm remains the same for each policy. In the case of initialization safety, the definitions are the different initialization blocks as shown in Figure 1, while the uses are statements which read a variable (i.e., contain an *rvar*). In the case of array bounds safety, the definitions correspond to the array declarations since the shadow variables get their values from the declared bounds, while the uses are statements which access an array variable.

The structure of the inference algorithm closely follows the outline in the previous section. The top-level function `ann_prog` (Figure 5) gets the safety property SP and the abstract syntax tree of the program P as arguments and returns the overall result by side-effects on P. It reduces the inference efforts by limiting the analysis to certain program hot spots which are determined by the so-called "hot variables" described in the next section.

`ann_prog` first accesses the property-specific patterns for definitions, uses and barriers. It then calls `compute_hotvars` to pass through the program and to collect all hot variables and hot uses, since annotations need to be constructed only for these. For each hot variable it then computes the abstracted CFG and iterates over all paths in the CFG that start with a hot use, before it finally constructs the annotations for the paths. This last step is broken into two functions `ann_def` and `ann_path` that are described in more detail in Section 4.4. Note that the hot variables are computed before the graph construction (and thus before the actual annotation phase), in order to minimize the work in that and subsequent stages.

```
proc ann_prog(SP:Property, P:AST) =
var patterns  : list Pattern;
    var       : Id;
    uses      : list Location;
    use       : Location;
    cfg       : CFG;
    path,rest : list Node;
    post      : Formula;
begin
  patterns := get_patterns(SP);
  foreach (var, uses) in compute_hotvars(SP, P) do
    cfg := compute_cfg(P, patterns, var);
    foreach use in uses do
      foreach path in compute_paths(cfg, use) do
        (post, rest):=ann_def(path);
        ann_path(var, use, rest, none, post);
end
```

**Figure 5.** Top-level Algorithm Structure

### 4.1 Hot Variables

As the VCG works its way backwards through the program, it gradually constructs a WSPC and generates VCs whenever required by the rules of the safety policy. These VCs will ultimately be discharged in the context of the safety substitutions that accumulate earlier in the program. If information about the content of a shadow variable is missing from that context, it must be provided by an annotation. Therefore, to figure out which annotations are required, we need to know at which points variables are used with "missing" information: we need a notion of availability.

We call a variable *available* (wrt. a safety property) at a program location if there are no barriers on the control flow paths from the variable's definition nodes to the use node, i.e., if this location is within reach of the variable's definition. For example, immediately after a scalar assignment, the assigned variable is available but it becomes unavailable if there is an intervening loop. We say that a variable use is *hot* if it unavailable, and call a variable a *hot variable* (or *hotvar* for short) if at least one of its uses is hot.

The function `compute_hotvars` used in Figure 5 maintains a list of available variables, initially set to empty, and scans forward through the program, deciding for each statement (and the given property) how it affects the availability of the variables. For example, we assume that scalar assignments add to the available variables, but array assignments do not: because arrays are typically accessed indirectly using loops and variable indices, all uses should be treated as hot. For each statement that matches the policy-specific use pattern, the algorithm also checks if the used variable is available; if it is not, that use is tagged as being hot.

The hot variables are approximated conservatively, i.e., we err on the side of designating uses as hot and could even treat *all* uses as hot. However, limiting the number of hot variables is an important optimization to cut down the number of graphs to be constructed (see Section 4.3).

### 4.2 Patterns and Pattern Matching

The algorithm uses patterns to capture the idiomatic code structures and pattern matching to find the corresponding code locations. Each pattern specifies a class of code fragments that are treated similarly by the algorithm, e.g., because they require a similar annotation.

The pattern language is essentially a tree-based regular expression language similar to XML-based languages like XPath [3]; Figure 6 shows its grammar. The language supports matching of tree literals $f(P_1, \ldots P_n)$ (if the signature $\Sigma$ is given by the programming language to be analyzed, we will also use its concrete syntax

$$
\begin{array}{rcll}
P & ::= & x & x \in X \\
 & | & f(P_1, \ldots, P_n) & f \in \Sigma \\
 & | & \texttt{\_} \mid P\texttt{?} \mid P\texttt{*} \mid P\texttt{+} & \\
 & | & P_1 \texttt{||} P_2 \mid P_1 \texttt{;} P_2 \mid P_1 \lhd P_2 & \\
 & | & P_1 \in P_2 \mid P_1 \notin P_2 &
\end{array}
$$

**Figure 6.** Pattern Grammar

to formulate example patterns), wildcards ( _) and the usual regular operators for optional (?), list (*) and non-empty list (+) patterns, as well as alternation ( || ) and concatenation (;) operators. $\lhd$ is a committed choice operator, which is similar to alternation, but tries the alternatives in a left-to-right order, and commits to the first match, i.e., does not backtrack into the other alternatives. The language also supports matching at arbitrary subterm positions (i.e., $P_1 \in P_2$ matches all terms that match $P_2$ and have at last one subterm that matches $P_1$; similarly, $P_1 \notin P_2$ matches all terms that match $P_2$ and have no subterm that matches $P_1$). Matching arbitrarily nested terms of the form $f(\cdots f(x) \cdots))$ is not required for our purposes.

However, the main difference from XPath and similar languages is that we use meta-variable patterns $x$ to introduce a limited degree of context dependency. Like a wildcard, an uninstantiated meta-variable matches any term but, unlike a wildcard, it becomes instantiated with the matched term and thus subsequently only in other instances of the instantiated pattern. For example, the pattern (_[_]:=_)+ matches the entire statement list A[1]:=1;A[2]:=2;B[1]:=1 while the pattern ($x$[_]:=_)+ matches only the two assignments to A but not the final assignment to B, due to the instantiation of $x$ with A. Further context-dependencies are introduced by multiple occurrences of the same meta-variable in a pattern. For example, a pattern of the form **for** $i$ := _ **to** _ **do** _[$i, i$]:= _ can be used to identify loops that access only the diagonal elements of any matrix.

The match procedure traverses terms first top-down and then left-to-right over the direct subterms. Meta-variables are instantiated eagerly (i.e., as close to the root as possible) but instantiations are undone if the enclosing pattern fails later on. List patterns follow the usual "longest match" strategy used in traditional regular expression matching. The match procedure returns as result a set of (*Location* $\times$ $\mathbb{N}$ $\times$ *Substitution*)-triples where the first two arguments are the root position and length of the match of the top-level pattern.

### 4.3 Abstracted Control Flow Graphs

The algorithm follows the control flow paths from variable use nodes backwards to all corresponding definitions and annotates the statements along these paths as required (see the next two sections for details). However, it does not traverse the usual control flow graphs but abstracted versions, in which entire code fragments matching specific patterns are collapsed into individual *nodes*. Since the patterns can be parameterized over the variables, separate abstracted CFGs must be constructed for each given hotvar. The construction is based on a straightforward syntax-directed algorithm as for example described in [16].[2] The only variation is that the algorithm first matches the program against the different patterns, using the algorithm described in the section above, and in the case of a match constructs a single node of the class corresponding to the successful pattern, rather than using the standard construction and recursively descending into the statements subterms.

In addition to *basic*-nodes representing the different statement types of the programming language, the abstracted CFG can thus contain nodes of several different pattern classes. The algorithm is based on the notions of *use*- and *def*-nodes and uses *barrier*-, *barrier-block*- and *block*-nodes as optimizations. All of these represent code chunks that the algorithm regards as opaque (to different degrees) because they contain no definition for the given variable. They can therefore be treated as atomic nodes for the purpose of path search, which drastically reduces the number of paths that need be explored. *barrier*-nodes represent any statements that require annotations, i.e., principally loops. They must therefore be re-expanded and traversed during the annotation phase of the algorithm. In contrast, *block*-nodes are completely irrelevant to the hotvar because they neither require annotations (i.e., contain no barriers) nor contribute to annotations (i.e., in our running example they contain no occurrence of the hotvar in an *lvar*-position). They can thus also remain atomic during the annotation phase, i.e., are not entered on path traversal. Blocks are typically loop-free sequences of assignments and (nested) conditionals. *barrier-blocks* constitute a further optimization by combining the other two concepts: they are essentially barriers wrapped into larger blocks. Hence, they must be re-expanded during annotation, like normal *barrier*-nodes. The algorithm must further distinguish between reaching a (barrier) block from behind and from within. Coming from behind, it can treat the block opaquely, as described above. Coming from within (i.e., starting from the initial use), the algorithm must ignore the block label, and regard the node as the underlying statement. This means it has to keep track of the previous location as it navigates along paths.

### 4.4 Annotation of Paths

For each hot use of a hot variable, the path computation in the previous section returns a list of paths to *putative* definitions. They have been identified by successful matches, but without the safety proof we cannot tell which, if any, of the definitions are relevant. In fact, it may be that several separate definitions are needed to fully define a variable for a single use. Consequently, all paths must be annotated. In a sense, the paths remain untrusted and trust is only established by annotating (more precisely, by the resulting VCs from) all barriers between the uses and definitions.

Paths are then annotated in two stages. First, unless it has already been done (during a previous path), the function `ann_def` used in Figure 5 annotates the definition at the end of the path and removes it from the rest of the path. If the use is contained within the definition then the path does not need to be continued because the definition will already have been fully annotated "internally", and the rest will be set to **nil**. Second, the definition's postcondition (which has to hold at the use location and along the path as well) is taken as the initial annotation and propagated along the path from the use back to the definition. Since this must take control flow into account, the current annotation is updated as the weakest precondition of the previous annotation. Both the computation of preconditions and the insertion of annotations are done node by node rather than statement by statement.

At each step, the path annotation function `ann_path` (see Figure 7) gets as arguments the hot variable, the original use location, the (rest of the current) path, the previous location, and the current weakest precondition. The previous location is needed to compute the precondition, and the hot variable and use location are used to prevent duplicate annotations. It first checks whether the current node is available. If so, or the current node is the last node before the definition, then since there are no more barriers the VCG will have all the information it needs from this point onwards and we

[2] Since the generators only produce well-structured programs, a syntax-directed graph construction is sufficient. However, we could, if necessary, replace the graph construction algorithm by a more general version that can handle ill-structured programs with arbitrary jumps.

```
proc ann_path(var:Id, use:Location, path:list Node,
              prv:Location, post:Formula) =
var cur,nxt : Location;
   node    : Node;
   nodes   : list Node;
   pre     : Formula;
begin
  case path of
    []            -> return
    [node|nodes] ->
      if available(node, nodes) or nodes = [_] then
        return
      else
        cur := get_location(node);
        nxt := get_location(head(nodes));
        if is_annotated(cur, post, use, var) then
          skip
        else
          if is_barrier(node) or is_opaque(node)then
            if within(prv, cur) then
              if is_loop(node) then
                if within(nxt, cur) then
                  ann_node_loop(node, post, use, var)
                else
                  ann_node_barrier(node, post, use, var)
              else
                skip
            else
              ann_node_barrier(node, post, use, var)
          else
            if is_loop(node) then
              if within(nxt, cur) then
                ann_node_loop(node, post, use, var)
              else
                ann_node_barrier(node, post, use, var)
            else
              skip;
        pre := node_wpc(prv, post, node);
        ann_path(var, use, nodes, cur, pre)
end
```

**Figure 7.** Path Annotation Algorithm

are finished. If not, we look to see if this node has already been annotated, and skip to the next node.

Otherwise, we distinguish several cases, depending on whether it is a loop or a barrier or an *opaque* node (i.e., a block or barrier-block), whether the previous node is contained within the current node, and whether the next node is within the current node. Once we have dealt with a node, the weakest precondition of that node is calculated, and we move on to the next node.

The WPC of a node is somewhat subtle and depends on whether or not it is a barrier or opaque, the statement itself (for basic blocks), and the previous location. In many cases the WPC does not change. For those cases where it does, the new WPC needs to be computed by looking at the statement. We distinguish atomic and compound statements. Compound statements (series, if, for, while) can only change the WPC if the previous location is after a loop, in which case $\texttt{statement\_wpc}(P, C) = \texttt{end}(C) \Rightarrow P$, where $P$ is the incoming postcondition, $C$ is the statement, and $\texttt{end}(C)$ is the termination condition for the loop, $C$. For **while** $b$ **do** $c$, this is $\neg\, b$, and for **for** $i$ `:=` $e_1$ **to** $e_2$ **do** $c$, it is $i > e_2$. In other words, the WPC says "if the loop has terminated then $P$". For atomic statements we compute the weakest precondition by calling the VCG (*without* generating safety conditions and substitutions) and simplifying the result.

### 4.5 Annotation of Nodes

The path traversal described above calls the actual annotation routines when it needs to annotate a node. Three classes of nodes need to be annotated: definitions, barriers, and basic nodes which are also loops.

The most important (and interesting) class is the definitions. This is really the core of the whole system, and where the annotation knowledge is represented in the form of *annotation schemas*, which take a match (identifying the pattern and location), and use meta-programming to construct and insert the annotations.

For example, each initialization block from Figure 1 is defined by a separate pattern and has a corresponding annotation schema. In each case, a final outer postcondition

$$\forall i \in \{1:N\} \cdot \forall j \in \{1:M\} \cdot x_{\text{init}}[i,j] = \text{INIT}$$

(where $x$ is the matrix) is inserted, while 1(b) and 1(c) also get an inner postcondition, as well as inner and outer invariants.

Note that even after a pattern has been successfully matched, an annotation schema might still fail its preconditions. For example, the binary assignment schema (Figure 1(a)) simply matches against a sequence of assignments, but the schema further requires that the indices of the first and last assignments are the low and the high, respectively.

The annotation schemas can handle more complicated examples than the "pure" definitions directly reflected by the patterns. A common situation is for a barrier to appear within a definition. Consider the following simple example:

```
for i:=1 to N do
  a[i]:=0;
  for j:=1 to M do ...
```

The definition pattern is a single nested initialization, but the inner **for**-loop means that an extra postcondition $a_{\text{init}}[i] = \text{INIT}$ is needed on the assignment to push the initialization through the body. However, if the **for**-barrier is *before* the assignment no extra annotation is needed. In general, the schemas are able to deal with such cases and maintain the "internal" flow of information within a definition.

## 5. Experiences

We have implemented the generic inference algorithm in about 4000 lines of Prolog code and instantiated it to certify initialization safety for code generated by AUTOBAYES and AUTOFILTER. The "declarative content" of the instantiation was surprisingly small: it only required instantiations of the pattern library but no changes to the algorithm itself.

### 5.1 AutoFilter

For AUTOFILTER, the definitions are given by two of the matrix initialization idioms in Figure 1, along with the direct matrix assignment operation **::=**. This is captured by the following pattern:

$$\begin{aligned} \mathit{def}_{AF}(x) ::= \; & x\texttt{:=\_} \;||\; x\texttt{::=\_} \\ ||\; & (x\texttt{[\_,\_]:=\_})\texttt{+} \\ ||\; & \textbf{for}\; i\; \texttt{:=\_}\; \textbf{to}\; \texttt{\_}\; \textbf{do} \\ & \quad \textbf{for}\; j\; \texttt{:=\_}\; \textbf{to}\; \texttt{\_}\; \textbf{do} \\ & \quad\quad \textbf{if}\; \texttt{\_}\; \textbf{then}\; x\texttt{[}i,j\texttt{]:=\_}\; \textbf{else}\; x\texttt{[}i,j\texttt{]:=\_} \end{aligned}$$

Like all patterns here, this is parametrized over a hotvar $x$, so that $\mathit{def}_{AF}(x)$ is the pattern of definitions for $x$, $\mathit{barrier}(x)$ (see below) is a barrier on a path from a use of $x$ to its definition, and so on. Note that $i$ and $j$ are "free" meta-variables that get instantiated by the actual loop index variables. The patterns can also contain "junk", i.e., arbitrary code that can be interspersed with the match. This is easily defined by a junk operator omitted here.

Barriers are defined as **for**-loops without any occurrence of the hotvar. Loops *with* the hotvar are then simply treated by the

| Spec. | $\mid P \mid$ | $\mid A \mid$ | $N$ | $T_{\text{gen}}$ | $T_{\text{ATP}}$ | $\mid A \mid$ | $N$ | $T_{\text{inf}}$ | $T_{\text{ATP}}$ |
|---|---|---|---|---|---|---|---|---|---|
| ds1 | 235 | 439 | 22 / - | 16 | 41 | 494 | 19 / - | 22 | 46 |
| iss | 523 | 441 | 27 / - | 29 | 52 | 547 | 24 / - | 46 | 49 |
| segm | 182 | 1278 | 105 / 6 | 22 | 628 | 1584 | 109 / - | 54 | 202 |
| | 178 | 1332 | 114 / 10 | 24 | 903 | 1643 | 108 / 5 | 54 | 556 |

**Table 1.** Annotation Generation vs. Annotation Inference

normal CFG-routines, i.e., not collapsed. Finally, blocks are either conditionals whose branches are deemed "irrelevant", which means they have no occurrence of a barrier or hotvar, or loops with an irrelevant body.

$$\begin{aligned} \mathit{barrier}_{AF}(x) &::= x \notin (\textbf{for } \_ := \_ \textbf{ to } \_ \textbf{ do } \_) \\ \mathit{block}_{AF}(x) &::= \textbf{if } (x \notin \_) \textbf{ then } \mathit{irr}(x) \textbf{ else } \mathit{irr}(x) \\ &\quad \mid\mid \textbf{for } \_ := \_ \textbf{ to } \_ \textbf{ do } \mathit{irr}(x) \end{aligned}$$

Here $\mathit{irr}(x) = (x \mid\mid \mathit{barrier}_{AF}(x)) \notin \_$ is an auxiliary pattern blocking all occurrences of the hotvar or a barrier. We omit the easy pattern for uses.

For the CFG construction, the above patterns are joined with the committed choice operator, i.e., the construction matches against the top-level pattern

$$\mathit{def}_{AF}(x) \lhd \mathit{barrier}_{AF}(x) \lhd \mathit{block}_{AF}(x).$$

Hence, the overlap between barriers and blocks is resolved deterministically.

### 5.2 AutoBayes

AUTOBAYES has similar patterns to AUTOFILTER, for vectors in addition to matrices, but does not need the **::=**-pattern since it does not generate direct matrix operations. It has several more **for**-loop patterns, as well as two additional language constructs, **abort**, which appears in the definition pattern, and **while**-loops, which can form additional barriers. Blocks and uses are defined in the same way as for AUTOFILTER, again extended to **while**-loops. Finally, for the CFG-construction, the patterns are again joined via committed choice.

$$\begin{aligned} \mathit{def}_{AB}(x) &::= (x[\_] := \_)+ \mid\mid (x[\_, \_] := \_)+ \\ &\quad \mid\mid \textbf{for } i := \_ \textbf{ to } \_ \textbf{ do } x[i] := \_ \\ &\quad \mid\mid \textbf{for } i := \_ \textbf{ to } \_ \textbf{ do } x[i, i \notin \_] := \_ \\ &\quad \mid\mid \textbf{for } i := \_ \textbf{ to } \_ \textbf{ do} \\ &\qquad \textbf{for } j := \_ \textbf{ to } \_ \textbf{ do } x[i, j] := \_ \\ &\quad \mid\mid \textbf{for } i := \_ \textbf{ to } \_ \textbf{ do} \\ &\qquad \textbf{for } j := \_ \textbf{ to } \_ \textbf{ do} \\ &\qquad\quad \textbf{if } \_ \textbf{ then abort else } x[i, j] := \_ \\ \mathit{barrier}_{AB}(x) &::= x \notin (\textbf{for } \_ := \_ \textbf{ to } \_ \textbf{ do } \_) \\ &\quad \mid\mid x \notin (\textbf{while } \_ \textbf{ do } \_) \\ \mathit{block}_{AB}(x) &::= \textbf{if } (x \notin \_) \textbf{ then } \mathit{irr}(x) \textbf{ else } \mathit{irr}(x) \\ &\quad \mid\mid \textbf{for } \_ := \_ \textbf{ to } \_ \textbf{ do } \mathit{irr}(x) \\ &\quad \mid\mid \textbf{while } \_ \textbf{ do } \mathit{irr}(x) \end{aligned}$$

### 5.3 Results

Table 1 compares the results achieved by the new algorithm to those previously achieved in the certifiable code generation approach. The first two examples are AUTOFILTER specifications. `ds1` is taken from the attitude control system of NASA's Deep Space One mission [28]. `iss` specifies a component in a simulation environment for the Space Shuttle docking procedure at the International Space Station. `segm` describes an image segmentation problem for planetary nebula images taken by the Hubble Space Telescope. For this, AUTOBAYES synthesizes two different versions of an iterative numerical clustering algorithm. For each example, the table lists the size $\mid P \mid$ of the generated program in lines of code, and then, for each approach, the sizes $\mid A \mid$ of the generated and inferred annotations, the numbers of generated and failed VCs, respectively, as well as the runtimes and proof times in seconds.

For the two AUTOFILTER examples, both techniques prove to be very similar. The inferred annotations are slightly larger (by 15–25%) than the generated ones but, due to simplifications, they induce fewer VCs. For both approaches, the programs are certifiable fully automatically: all VCs are proven by the ATP. For the AUTOBAYES example, the situation is more complicated. Here, the previous approach to annotation generation within the code generator has not kept up with ongoing development and the annotations are now insufficient to prove the programs safe—even though they are. With the patterns described above, annotation inference can, in contrast, certify the first program but it too remains too weak for the second program, as a required code pattern turns out to be missing. However, this pattern could be easily added, and with significantly less effort than modifying the generator itself. In both cases, the inferred annotations are again slightly larger, with fewer VCs induced.

Since it needs to build and traverse the CFGs, the inference approach is (substantially) slower than the generation approach, which only needs to expand templates. However, the introduction of *block*- and *barrier*-nodes cuts down the size of the CFGs dramatically, and we expect further speed-up from an optimized implementation. Moreover, the limiting factors overall are the proof times which are comparable (modulo failed VCs) in all cases, indicating that the inference does not introduce new complexity for the ATP.

## 6. Related Work

Logical annotations were recognized early on as one of the bottlenecks in program verification. Wegbreit [26] complained that "completely specifying the predicates on loops is tedious, error prone and redundant", and claimed that "loop predicates can be derived mechanically". Like other early work [11, 17], his approach is based on predicate propagation. Such methods use inference rules similar to a strongest postcondition calculus to push an initial logical annotation forward through the program. Loops are handled by a combination of different heuristics like weakening or strengthening and loop unrolling, until a fixpoint is achieved. However, these methods still need an initial annotation, and unlike our approach, the loop handling still induces a search space at inference time. Moreover, the constructed annotations are often only candidate invariants and need to be validated (or refuted) during inference, because they increase the search space.

Abstract interpretation has been used to infer annotations in separation logic for pointer programs [18] although the techniques required there are fairly specialized and elaborate compared to our patterns. The Coverity static analyzer [1] can be customized by macros that are simple versions of our patterns.

Finally, generate-and-test methods have been applied to our problem. Here, the generator phase uses a fixed pattern catalogue to construct candidate annotations while the test phase tries to validate (or refute) them, using dynamic or static methods. Daikon [12] is the best-known dynamic annotation inference tool in this category. Its tester accepts all candidates that hold without falsification but with a sufficient degree of support over the test suite. In order to verify the candidates, Daikon has also been combined [21] with the ESC/Java static checker [15]. In some cases, this combination even resulted in full safety proofs (wrt. the safety policy supported by ESC/Java). In general, however, dynamic annotation generation techniques remain incomplete because they rely on a test suite to generate the candidates and can thus miss annotations on paths that are not executed often enough (or not at all). Houdini [14] is a static

generate-and-test tool that uses ESC/Java to statically refute invalid candidates. Since ESC/Java is a modular checker, Houdini has to start with a candidate set for the entire program and then iterate until a fixpoint is reached. This increases the computational effort required, and in order to keep the approach tractable, the pattern catalogue is deliberately kept small. Hence, Houdini is incomplete, and acts more as a debugging tool than as a certification tool.

## 7. Conclusions and Future Work

The certification system based on annotation inference as described here is much more flexible and extensible than the previous certification architecture [7]. Over time, extensions and modifications to the code generators had led to a situation of "entropic decay" where the generated annotations had not kept pace with the generated code. The new inference mechanism was able to automatically certify the same programs as the old system, as well as some subsequent extensions. However, as Table 1 shows, the re-construction is not yet complete, and we continue to extend the new system. These system extensions require less effort than before since the patterns and annotation schemas are expressed declaratively and in one place, in contrast to the previous decentralized architecture where certification information is distributed throughout the code generator. Identifying the patterns was an iterative process. We were helped in this by a browser [9] which allows tracing between VCs and statements of the auto-generated code. This let us pinpoint missing annotations more easily and, thus, determine missing patterns.

Our approach offers a general framework for augmenting code generators with a certification component, and we have started a project to apply it to MathWorks Real-Time Workshop [2]. Our techniques could also be adapted to other annotation languages.

There is a strong interaction between the VCG and the annotations. It is possible to modify the VCG so that it does some analysis and requires less annotations. This would, however, mean that a greater part of the certification system must be trusted. Nevertheless, we would like have a "safety dial" whereby users can trade off trustedness with speed (which depends, *inter alia*, on the number of annotations which must be checked). Further empirical studies will be required to determine the most effective balance. However, we have already implemented several optimizations which cut down on redundant annotations. This is important since the same annotations can arise on multiple paths. Furthermore, many computational optimizations could be achieved by merging several of the phases.

Currently, the entire variability over the set of programs generated by AUTOFILTER and AUTOBAYES can be captured by the fixed set of patterns used. In general, this is not necessarily the case. However, then the code generator could be extended to generate *annotation plans* which would, as an extension to the techniques presented in this paper, supply additional program-specific patterns, and would also allow the default inference algorithm to be modified. This could further increase the applicability of our techniques.

## References


[1] `www.coverity.com`.

[2] `www.mathworks.com/products/rtw/`

[3] XML Path Language (XPath) Version 1.0, 1999. `www.w3.org/TR/xpath`.

[4] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming*. Addison-Wesley, 2005.

[5] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[6] E. Denney and B. Fischer. Correctness of source-level safety policies. In *FM'03*, *LNCS 2805*, pp. 894–913. Springer, 2003.

[7] E. Denney and B. Fischer. Certifiable program generation. In *GPCE'05*, *LNCS 3676*, pp. 17–28. Springer, 2005.

[8] E. Denney, B. Fischer, and J. Schumann. Adding assurance to automatically generated code. In *8th Intl. Symp. High-Assurance Systems Engineering*, pp. 297–299. IEEE Press, 2004.

[9] E. Denney and B. Fischer. A program certification assistant based on fully automated theorem provers. In *Intl. Workshop User Interfaces for Theorem Provers*, pp. 98–116, April 2005.

[10] E. Denney, B. Fischer, and J. Schumann. An empirical evaluation of automated theorem provers in software certification. *Intl. J. of AI Tools*, 15(1):81–107, 2006.

[11] N. Dershowitz and Z. Manna. Inference rules for program annotation. *ICSE-3*, pp. 158–167. IEEE Press, 1978.

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

[13] B. Fischer and J. Schumann. AutoBayes: A system for generating data analysis programs from statistical models. *J. Functional Programming*, 13(3):483–508, 2003.

[14] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME'01*, *LNCS 2021*, pp. 500–517. Springer, 2001.

[15] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI'02*, pp. 234–245. ACM Press, 2002.

[16] M.J. Harrold and G. Rothermel. Syntax-directed construction of program dependence graphs. Technical Report OSU-CISRC-5/96-TR32, The Ohio State University, 1996.

[17] S. Katz and Z. Manna. Logical analysis of programs. *CACM*, 19(4):188–206, 1976.

[18] O. Lee, H. Yang, and K. Yi. Automatic Verification of Pointer Programs Using Grammar-Based Shape Analysis. In *ESOP'05*, *LNCS 3444*, pp. 124–240. Springer, 2005.

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

[20] G. C. Necula. Proof-carrying code. In *POPL-24*, pp. 106–19. ACM Press, 1997.

[21] J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected invariants: Integrating Daikon and ESC/Java. In *First Workshop on Runtime Verification*, *Elec. Notes in Theoretical Computer Science*, 55(2). Elsevier, 2001.

[22] C. Rich and L. M. Wills. Recognizing a program's description: A graph-parsing approach. *IEEE Software*, 7(1):82–89, 1990.

[23] D. R. Smith. KIDS: A semi-automatic program development system. *IEEE TSE*, 16(9):1024–1043, 1990.

[24] M. Stickel et al. Deductive composition of astronomical software from subroutine libraries. In *CADE-12*, *LNAI 814*, pp. 341–355. Springer, 1994.

[25] I. Stürmer and M. Conrad. Test suite design for code generation tools. In *ASE-18* pp. 286–290. IEEE, 2003.

[26] B. Wegbreit. The synthesis of loop predicates. *CACM*, 17(2):102–112, 1974.

[27] M. Whalen, J. Schumann, and B. Fischer. Synthesizing certified code. In *FME'02*, *LNCS 2391*, pp. 431–450. Springer, 2002.

[28] J. Whittle and J. Schumann. Automating the implementation of Kalman filter algorithms. *ACM Trans. Mathematical Software*, 30(4):434–453, 2004.