A High-Level Certification Language for 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

Program verification using Hoare-style techniques requires many logical annotations. We have previously shown that a generic annotation inference algorithm can be used to weave in all annotations required to certify safety properties for automatically generated code. The algorithm is implemented as part of our AUTOCERT system. It uses patterns to capture generator- and property-specific code idioms and property-specific meta-program fragments to construct the annotations. It is customized by specifying the code patterns and integrating them with the meta-program fragments for annotation construction. However, the latter part has so far involved tedious and error-prone low-level term manipulations, which has made customization difficult.

Here, we describe an *annotation schema compiler* that simplifies and largely automates this customization task. It takes a collection of high-level declarative *annotation schemas* tailored towards a specific code generator and safety property, and generates all glue code required for interfacing with the generic algorithm core, thus effectively creating a customized annotation inference algorithm. The compiler raises the level of abstraction and simplifies schema development and maintenance. It also takes care of some more routine aspects of formulating patterns and schemas, in particular handling of irrelevant program fragments and irrelevant variance in the program structure, which reduces the size, complexity, and number of different patterns and annotation schemas that are required. This paper further contributes to developing a declarative and generative approach to logical annotations. The improvements described here make it easier and faster to customize the system to a new safety property or a new generator. We have been able to show different properties for a variety of programs generated by our AUTOBAYES and AUTOFILTER generators. We have also applied AUTOCERT to code derived from MathWorks Real-Time Workshop, and show some initial results.

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

The verification of program safety and correctness using Hoare-style techniques requires large numbers of logical annotations (principally loop invariants, but also pre- and post-conditions) that must be woven into the program. These annotations constitute cross-cutting concerns, which makes it hard to generate the annotations in parallel with the code. For example, verifying even a single array access safe may need annotations throughout the entire program to ensure that all the information about the array and the indexing expression that is required for the proof is available at the access location.

In previous research [8], we have shown that for the case of certifying safety properties for automatically generated code, a generic annotation inference algorithm can be applied to weave the annotations into the program. The algorithm uses techniques similar to aspect-oriented programming and exploits the idiomatic structure of automatically generated code and the relative simplicity of safety properties. It uses patterns to capture the generator- and property-specific code idioms and property-specific meta-program fragments associated with these patterns to construct the annotations. This allows us to handle domain-specific properties more easily than certification approaches based on abstract interpretation. The algorithm is implemented as part of our AUTOCERT system for the safety certification of automatically generated code. It constructs an abstracted control-flow graph (CFG), using the patterns to collapse the code idioms into single nodes. Then, it traverses this graph and follows all paths from use-nodes backwards to all corresponding definitions. The algorithmic core of AUTOCERT (i.e., CFG construction and transversal) is fully generic. It is customized for a given code generator and safety property by specifying the code patterns and integrating them with the implementation of the meta-program fragments for annotation construction. However, while the former part can build on a clean, declarative pattern language, the latter part has so far involved tedious and error-prone low-level term and program manipulations. This has made it more difficult than necessary to customize and extend our system and has slowed down our progress.

Here, we describe an *annotation schema compiler* that simplifies and largely automates this customization task, in continuation of our general line of research. It takes a collection of *annotation schemas* tailored towards a specific code generator and safety property, and generates all glue code required for interfacing with the generic algorithm core, thus effectively creating a customized annotation inference algorithm. The compiler allows us to represent

all the knowledge required to handle a class of specific certification situations declaratively and in one central location (i.e., in the annotation schemas), which raises the level of abstraction and simplifies development and maintenance. The compiler also takes care of some more routine aspects of formulating patterns and schemas, in particular handling of irrelevant program fragments ("junk") and irrelevant variance in the program structure (e.g., the order of branches in conditionals), which reduces the size, complexity, and number of different patterns and annotation schemas that are required. Taken together, the improvements described here make it easier and faster to customize the generic annotation inference algorithm to a new safety property or a new generator.

In this paper, we thus build on but substantially improve over our previous work on annotation inference for automatically generated code [8]. We make new technical, empirical, and methodological contributions. Technically, our main contributions are the development of the schema compiler and the implicit junk handling by the compiler. Compared to [8], we have also extended the pattern language by additional constraint operators, which makes it more expressive and allows more context-sensitivity in the patterns, thus minimizing reliance on the use of arbitrary meta-programming functionality in the guards. Empirically, our main contribution here is a significantly extended evaluation of our general annotation inference approach. Based on the extensions described here, we were able to show several safety properties (initialization-before-use, array bounds, and matrix symmetry) for a variety of programs generated by our AUTOBAYES [12] and AUTOFILTER [31] generators. We have also started to apply AUTOCERT to some code derived from Real-Time Workshop, and we report on the first experiences we have gained so far. Methodologically, this paper is a further contribution to developing a declarative and generative approach to logical annotations. It is also a step towards our ultimate goal, developing a programmable certification language for automated code generators.

The remainder of this paper is organized as follows. The next section gives some general background on the safety certification of automatically generated code. Section 3 then summarizes the underlying annotation inference algorithm as far as is required here; more details can be found in [8]. Section 4 contains a description of the different aspects of the annotation schema compiler, while Section 5 focuses on the practical experience we have gained so far. The final two sections discuss related work and conclude with an outlook on future work.

2. General Background

In this section we briefly summarize our approach to safety certification of automatically generated code. We split the certification problem 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 fully automatically prove that the code satisfies the required properties. We further simplify the problem by restricting our attention to the certification of specific safety properties, rather than arbitrary functional requirements. We then exploit both the highly idiomatic structure of automatically generated code and the restriction to specific safety properties to automate the annotation generation as well. Since generated code only constitutes a limited subset of all possible programs, no new "eureka" insights are required at annotation inference time. Since safety properties are simpler than functional correctness, the required annotations are also simpler and more regular.

Safety Certification The purpose of safety certification is to demonstrate that a program does not violate certain conditions during its execution. A *safety property* [6] 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. In this paper, we focus on three example properties, variable initialization before use (*init*), array bounds (*array*), and matrix symmetry (*symm*). *init* ensures that each variable or individual array element has been explicitly assigned a value before it is used. *array* requires each access to an array element to be within the specified upper and lower bounds of the array. Both are typical examples of language-specific properties, but our approach can be used with other, domain-specific properties such as matrix symmetry as well. *symm* ensures that the covariance matrices that are manipulated by the programs generated by AUTOFILTER remain symmetric. See [6, 7, 8] for more details and the rules of different safety policies.

VC Processing and Annotations As usual in Hoare-style verification, a VCG traverses annotated code and applies the calculus rules of the safety policy to produce verification conditions (VCs). These are then simplified, completed by an axiomatization of the relevant background theory and passed to an off-the-shelf ATP. If all VCs are proven, we can conclude that the program is safe with respect to the safety policy, and, given the policy is sound, also the safety property. Note that the annotations only serve as "hints" or lemmas that must be established in their own right. Consequently, they can remain untrusted—a wrong annotation cannot compromise the assurance provided by the system.

Idiomatic Code Automated code generators derive lower-level code from higher-level, declarative specifications, usually by combining a finite number of building blocks (e.g., templates, components, or schemas) following a finite number of combination methods (e.g., template expansion, component composition, or schema application). This usually results in highly *idiomatic code*, i.e., code that exhibits some regular structure beyond the syntax of the programming language and that uses similar constructions for similar problems. For example, Figure 1 shows the example of the three matrix initialization idioms employed by AUTOFILTER and AUTOBAYES. The idioms are essential to our approach because they (rather than the building blocks or combination methods) determine the interface between the code generator and the inference algorithm. For each generator and each safety property, our approach thus requires a customization step in which the relevant idioms are identified and formalized as patterns. 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};
...
A[1,m]:= a_{1,m};
A[2,1 ]:= a_{2,1};
...
A[n,m]:= a_{n,m};
```
(a)

```
for i:= 1 to n do
  for j:= 1 to m do
    B[i,j]:= b;
```
(b)

```
for i:= 1 to n do
  for j:= 1 to m do
    if i=j then
      C[i,j]:= c
    else
      C[i,j]:= c';
```
(c)

Figure 1. Idiomatic matrix initializations in AUTOBAYES and AUTOFILTER

System Architecture Figure 2 shows the overall architecture of our AUTOCERT certification system. 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 a certification subsystem, consisting of the standard components of Hoare-style verification approaches, i.e., VCG, simplifier, ATP, proof checker, and domain theory.

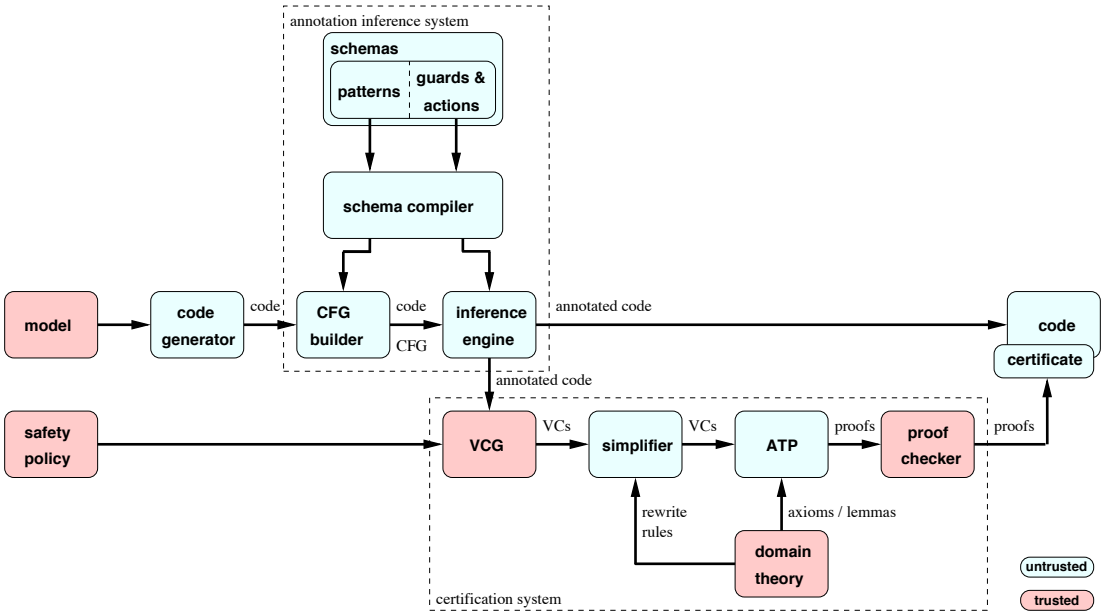

Figure 2. AUTOCERT system architecture

These components and their interactions are described in more detail in [6, 9, 30]. As in the proof-carrying code approach [23], the architecture distinguishes between trusted and untrusted components, shown in Figure 2 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 entire annotation inference subsystem (including the pattern library and annotation schemas) also remain untrusted since the resulting annotations simply serve as lemmas for the subsequent analysis steps.

3. Annotation Inference Algorithm

The key knowledge that drives the annotation construction is the set of idiomatic coding patterns which are used in a program. However, we need to distinguish different classes of patterns, in particular, definitions, uses, and barriers. *Definitions* describe idioms that establish the safety property of interest for a given variable, while *uses* refer to locations where the property is required. *Barriers* represent any statements that require annotations, i.e., principally loops. These patterns are specific to the given safety property, but the algorithm remains the same for each property. In the case of initialization safety, the definitions are the different initialization blocks, 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 fragments which set the values of array indices, while the uses are statements which access an array variable. In both cases, barriers are loops.

The inference algorithm itself is then based on two related key observations. First, it is sufficient to annotate only in reverse along all CFG-paths between uses (where the property is required) and definitions (where it is established). Second, along each path it is sufficient to annotate only with the definition's post-condition, or more precisely, the definition's post-condition under the weakest pre-condition transformation that is implemented in the VCG.

The top-level function of the inference algorithm builds and traverses the CFG and returns the overall result by side-effects on the underlying program 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 [8]. 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 the subsequent stages. For each hot variable the algorithm 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.

3.1 Abstracted Control Flow Graphs

The algorithm follows the control flow paths from variable use nodes backwards to all corresponding definitions and annotates the barrier 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 hot variables, separate abstracted CFGs are constructed for each given hot variable. The construction is based on a straightforward syntax-directed algorithm as for example described in [17].[1] The only variation is that the algorithm first matches the program against the different patterns, 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

[1] 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.

contain nodes of the different pattern classes. The algorithm is based on the notions of the *use*- and *definition*-nodes and uses *barrier*-, *barrier-block*- and *block*-nodes as optimizations. The latter three 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 hot variable 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 hot variable 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.

3.2 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.

Paths are then annotated in two stages. First, unless it has already been done during a previous path, the definition at the end of the path is annotated. Second, the definition's post-condition (which has to hold at the use location and along the path as well) is taken as the initial annotation and propagated back along the path from the use to the definition. Since this must take computations and control flow into account, the current annotation is updated as the weakest pre-condition of the previous annotation. Both the computation of pre-conditions and the insertion of annotations are done node by node rather than statement by statement.

3.3 Annotation of Nodes

The path traversal described above calls the actual annotation routines (whether implemented manually or generated from the annotation schemas) when it needs to annotate a node. Three classes of nodes need to be annotated: definitions, barriers, and basic nodes which are also loops. However, the most important (and interesting) class is the definitions.

The definitions comprise the core of the whole system because their annotations (more precisely, their final post-conditions) are used as initial values for annotation along the paths. For example, for each of the three different initialization blocks shown in Figure 1, a separate annotation schema can be defined, which in each case inserts a final (outer) post-condition

$$\forall i, j \cdot 1 \leq i \leq N \wedge 1 \leq j \leq M \Rightarrow x_{\text{init}}[i, j] = \text{INIT}$$

establishing that the matrix x is initialized. However, the schemas also need to maintain the "internal" flow of information within a definition. Hence, the schemas dealing with the situations shown in Figure 1(b) and 1(c) also need to insert an inner post-condition, as well as inner and outer loop invariants.

Note that even after a pattern has been successfully matched, an annotation schema might still fail its guards. For example, the schema handling the binary assignment idiom in Figure 1(a) simply matches against a sequence of assignments, but requires that the indices of the first and last assignments are the lower and upper bound of the array, respectively.

$$
\begin{array}{rcll}
P & ::= & x & x \in X \\
 & | & f(P_1, \ldots, P_n) & f \in \Sigma \\
 & | & \_ \mid P\texttt{?} \mid P\texttt{*} \mid P\texttt{+} \mid P_1 \ \texttt{...}\ P_2 & \\
 & | & P_1 \ \texttt{;}\ P_2 \mid P_1 \ \texttt{||}\ P_2 \mid P_1 \ \texttt{<+}\ P_2 & \\
 & | & P_1 \ \texttt{//}\ P_2 \mid P_1 \setminus\!\setminus P_2 \mid P_1 \supset P_2 \mid P_1 \not\supset P_2 \mid P_1 \notin P_2 & \\
 & | & I & \\
I & | & P \leftarrow U \mid P \ \texttt{@}\ x & \\
U & ::= & \texttt{\&(}A\ \{\texttt{,}\ A\}\texttt{)} \mid \texttt{\&\&(}A\ \{\texttt{,}\ A\}\texttt{)} \mid \langle \textit{prim-op} \rangle & \\
A & ::= & \textbf{inv}\ F \mid \textbf{pre}\ F \mid \textbf{post}\ F & F \in \mathcal{F}
\end{array}
$$

Figure 3. Grammar of annotated patterns

4. Annotation Schema Representation and Compilation

An *annotation schema* is a declarative representation of the knowledge required to handle a class of specific certification situations. Its main components are a code pattern that describes both the structure of the object program fragments to which the schema is applicable and where the annotations will be added, and two lists of run-time guards and actions that will be first executed when the pattern is matched against the object program, and then used to compute the actual annotations that are added. While the patterns are formulated in the extended pattern language described in Section 4.1, the guards and actions are simply arbitrary code fragments in the underlying meta-programming language (in our case, Prolog), eliminating the need for a dedicated action language.

The annotation schema compiler takes a collection of annotation schemas tailored towards a specific code generator and safety property, and compiles it down into a customized annotation inference algorithm. Since we are reusing the core annotation inference algorithm outlined above and described in more detail in [8], which is implemented in Prolog, the output of the compiler is simply a set of Prolog clauses. Since the annotation schema compiler is implemented in Prolog as well, annotation schemas can simply be represented by Prolog facts or clauses, as shown for example in Figure 4. The representation of schemas is explained in more detail in Section 4.2 while the schema compiler itself is described in Section 4.5. Sections 4.3 and 4.4 describe pattern pre-processing and refinement, respectively.

4.1 Extended Pattern Language

The annotation inference algorithm uses patterns to capture the idiomatic code structures and pattern matching to find the corresponding code fragments and build the CFG. The annotation schemas use an extended version of these patterns that also provides additional operators to support the interaction with the meta-program fragments which construct the annotations.

The pattern language is essentially a tree-based regular expression language similar to XML-based languages like XPath [2]; Figure 3 shows its grammar. The language supports matching of tree literals $f(P_1, \ldots P_n)$ over a given signature Σ, wildcards (`_`) and the usual regular operators for optional (`?`), list (`*`) and non-empty list (`+`) patterns, as well as alternation (`||`) and concatenation (`;`) operators. `...` is an ellipsis operator, which allows the concise formulation of enumerations. P_1 `...` P_2 is compiled into P_1`;`P`*;`P_2, where $P = \text{lcs}(P_1, P_2)$ is the least common subsumer (or anti-unifier) of P_1 and P_2. This is computed by replacing any two different subterms at corresponding positions in the two terms by a fresh variable. `<+` 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.

Unlike a "pure" regular expression language, our pattern language allows us, to some limited degree, to express context dependencies. This can be achieved by two different mechanisms, constrained patterns and pattern meta-variables. Constrained patterns generalize the idea of lookahead that is well-known from regular expression matching. A *constrained pattern* P_1 *op* P_2 consists of a base pattern P_1 that must be matched against the input, and will eventually be returned as match result, and a constraining pattern P_2 that can rule out potential base matches, depending on the given constraint operator *op*. Possible constraint operators are lookahead (`//`) and its complement (`\\`), which only check the right siblings of the term matched against the base pattern (i.e., work horizontally), and various forms of subterm matching, which only check its descendants and ancestors (i.e., work vertically). Hence, $P_1 \supset P_2$ matches all terms that match P_1 and have at last one subterm that matches P_2; similarly, $P_1 \not\supset P_2$ matches all terms that match P_1 and have no subterm that matches P_2.[2] In contrast to the inward-looking operators $\supset$ and $\not\supset$, the $\notin$-operator looks outward: $P_1 \notin P_2$ checks for instances of P_1 which are not within any enclosing occurrence of P_2. This has proved very useful to rule out accidental matches. Uninstantiated *pattern meta-variables* match any term but, unlike a wildcard, they then become instantiated with the matched term and subsequently match only against further instances of the first match. 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`.

Another extension of the pattern language describes interactions I with the meta-program fragments constructing the actual annotations. The two operators $\leftarrow$ and `@` supporting this interaction will be stripped away before the pattern is used for matching purposes, but they are used to compile the guards and actions of the corresponding schema. The weave-operation $P \leftarrow U$ executes an update action U on the program fragment matched against P when the annotation schema is applied, and thus weaves in the annotation. U can be an arbitrary meta-program operation *prim-op* of type $T_\Sigma \rightarrow T_\Sigma$, but typically it just adds a list of annotations to the target fragment, and we provide two built-in operations for this case. `&(`A`)` simply adds the annotations A to the target fragment, while `&&(`A`)` recursively adds A to all barriers inside the target fragment. This is mostly used for the "junk" handling described in Section 4.3. In both cases, annotations are simply formulas $F \in \mathcal{F}$, labeled with their purpose as invariant, pre- or post-condition. The access-operator P `@` x binds the variable x to the term matched against P, so that it can be referred to in the guards and actions. This is similar to the use of pattern meta-variables, but allows P to be further instantiated.[3]

The match procedure traverses terms first top-down and then left-to-right over the direct subterms, returning as result triples where the first two arguments are the root position and length of the match of the top-level pattern, and the third is a substitution with bindings for the pattern meta-variables. The 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. Lookahead and subterm matching are implemented in a straightforward way, but the performance of the pattern matcher has been sufficient so far.

The match procedure also supports a limited form of matching modulo theory: users can specify how tree literal patterns can be mapped onto terms.[4] We use this to handle some irrelevant syntactic variance in the programs, for example, to identify block patterns of the form {`*;`P`;*`} with single statements matching P. This feature has proved very useful, but it has to be used with care, since the indiscriminate use of such mappings can increase the search space for matching substantially and can also lead to unintended matches and hence a loss of control.

[2] In [8], these were denoted by $P_2 \in P_1$ and $P_2 \notin P_1$, respectively.

[3] This feature is similar to the layered pattern matching using the as-operator in SML.

```
schema(for_assign
, SP
, def(A)
, (for (I := _ to _)@ Index do
     ((A[*⊅ I; I; *⊅ I])@ AI := _ ⊅ A) ← &(post PostAI)
  ) ← &(inv Inv, post Post)
, default
, []
, [anngen_upd(SP, AI, PostAI),
   anngen_for(SP, AI, Index, Inv, Post)]
) :- SP=init ; SP=range(_).
```

Figure 4. Annotation schema `for_assign`

4.2 Schema Representation

An annotation schema bundles together all knowledge that is required by the annotation inference algorithm to handle a class of specific certification situations. In addition to the pattern and the run-time guards and actions this also includes the safety policy or policies under which the schema is applicable and the node class that will be attached to the matched object program fragments. Since our schema compiler is implemented in Prolog, we simply represent schemas by Prolog facts or clauses. This allows us to use arbitrary Prolog code as compile-time guards and actions to the schemas and thus to further simplify their formalization. In the example shown in Figure 4,[5] we can thus use the same schema (with appropriately parameterized actions) for two of the different safety properties *init* and *range* (a vector satisfies *range*(`dim`(A, N)) if all its entries are within the bounds of the Nth dimension of array A), although we concentrate on *init* here. The schema clauses also contain some additional information that is used by the schema compiler, namely the schema name (for reference purposes), and the name of a pattern pre-processing predicate (here `default`) which can be used to simplify the description of the patterns and the advice. See Section 4.3 for details.

The `for_assign` schema shown in Figure 4 is designed to annotate loops that initialize arrays element by element. For example, in order to facilitate a proof that

for i **:=** 1 **to** N **do**
  a**[**i**]** **:=** b**[**i**];**

actually initializes the array `a`, the schema needs to construct an appropriate loop invariant and post-condition, resulting in the annotated loop

for i **:=** 1 **to** N **inv** $\forall\, j \cdot\ 1 \leq\ j < i\ \Rightarrow a_{\text{init}}[j] = \text{INIT}$ **do**
  a**[**i**]** **:=** b**[**i**];**
  post $a_{\text{init}}[i] = \text{INIT}$
post $\forall\, j \cdot\ 1 \leq\ j \leq\ N\ \Rightarrow a_{\text{init}}[j] = \text{INIT}$

[4] Our implementation is not based on full matching modulo equational theories as for example used in Maude [4]. It is thus incomplete for certain theories.

[5] Here, and in the rest of the paper, we type-set the patterns using concrete syntax to improve the legibility of the schemas. Our implementation uses standard Prolog terms, but supporting concrete syntax would be easy [13].

The first step in designing this schema is to specify the core pattern that will be used to identify instances of the general loop structure in the program. Here we are looking for single **for**-loops with arbitrary lower and upper bounds, where the loop body consists of an update of an arbitrary array A, in which the loop's index variable I is used as index. We allow additional indices left and right of I, provided they contain no further occurrences of I (thus restricting the schema to arrays effectively used as vectors), and require that the right-hand side of the assignment contains no further occurrences of the array A that is being initialized. This can be expressed concisely in our pattern language:

for I **:=** _ **to** _ **do**
  A[*∌ I ; I ; *∌ I] **:=** _ ∌ A

The second step is to add ←-operations to splice the constructed annotations into the appropriate locations. As outlined above, we need an invariant *Inv* and post-condition *Post* on the loop itself. However, we also need to specify a post-condition *PostAI* on the individual array-update, which will be used by the pattern pre-processing described following section. This yields

(**for** I **:=** _ **to** _ **do**
  (A[*∌ I ; I ; *∌ I] **:=** _ ∌ A) ← &(**post** *PostAI*)
) ← &(**inv** *Inv*, **post** *Post*)

The third and final step is to add the guards and the actions that actually construct the annotations. Here, guards are not required and the actions consists of calls to two predicates provided by our meta-programming kernel. `anngen_upd` and `anngen_for` construct the post-condition for a single array-update and the loop invariant and post-condition, respectively. Both predicates require access to specific parts of the actual program fragment matched against the pattern, in particular the complete left-hand side of the array-update. Since this is not bound by a pattern meta-variable—note that A only contains the name of the array, not the entire access—the pattern used in the schema contains additional variables like AI that are bound to the relevant subterms and then used to pass them into the predicates (see Figure 4). In the above example, we thus get the annotations $PostAI \equiv a_{\text{init}}[i] = \text{INIT}$, $Inv \equiv \forall j \cdot 1 \leq j < i \Rightarrow a_{\text{init}}[j] = \text{INIT}$, and $Post \equiv \forall j \cdot 1 \leq j \leq N \Rightarrow a_{\text{init}}[j] = \text{INIT}$, as expected.

The use of specialized meta-programming functionality such as `anngen_upd` could be considered to be somewhat non-declarative, since it requires understanding of a potentially large and complicated meta-programming kernel. However, in our opinion (and experience), this is unavoidable to achieve genericity: unless we severely limit the range of safety policies, we need some flexible mechanism to construct the core annotations. The advantage of using the schema compiler is that it minimizes the user's exposure to this, and helps separating out the fully declarative aspects (i.e., the pattern) from the "less declarative" aspects (i.e., the guards and actions). Also, since these functions encapsulate general induction principles, very few of them are needed.

4.3 Pattern Pre-processing

Often, even auto-generated code does not exactly fit the pattern specified in a schema, but contains additional but irrelevant "junk" statements, i.e., statements that are irrelevant to the current hot variable. Such junk can be part of the original program structure, or it can be introduced by optimizations (e.g., loop-invariant computations that are hoisted out of an inner loop). Consider for example the `for_for_assign` schema shown in Figure 5, which can be used to annotate doubly nested **for**-loops initializing a single matrix A. But this schema should also apply in situations where the outer loop contains additional statements before or after the inner

```
schema(for_for_assign
, init
, def(A)
, (for (I := _ to _)@ IndexI do
     (for (J := _ to _)@ IndexJ do
        ((A[(I;J) ⇚ (J;I)])@ AI := _ ∌ A) ← &(post PostAI)
     ) ← &(inv InvJ, post PostJ)
  ) ← &(inv InvI, post PostI)
, default
, []
, [anngen_upd(init, AI, PostAI),
   anngen_for(init, AI, IndexI, IndexJ,
              InvI, PostI, InvJ, PostJ)]
).
```

Figure 5. Annotation schema `for_for_assign`

loop, and similarly for the inner loop, e.g., if, as the result of a loop fusion, two matrices are initialized at the same time.

Extending the schema to cover these cases is a two-staged process. First, the junk statements need to be "matched away", which can be achieved by adding list wildcards to the arguments of the statement patterns. Some care must be taken to ensure that these do not conflict with the proper pattern; we thus add additional constraints to the list wildcards (see Figure 6). However, the junk fragments can also contain statements that match barrier patterns and thus require annotations as well. These fragments will not be annotated during the CFG traversal because they have become part of the definitions. Consequently, the junk fragments must in the second stage be annotated by the definition schema as well.

The entire process can be automated because the annotations required for the different junk positions can be derived systematically from the annotations given in the original pattern using the notion of *current annotation*:

- On entry to a loop pattern, the current annotation is set to the invariant attached to the loop (or to true, if no invariant is given), and its old value is saved.
- On exit from a loop pattern, the current annotation is restored to the saved value, and the post-condition attached to the loop (if any) is added to it.
- For any other pattern, the attached post-condition (if any) is added to it.

The current annotation is then used to annotate any barriers that are contained in the junk fragments. The annotation schema compiler simply keeps the current annotation while it pre-processes the patterns, and whenever it inserts a list wildcard to match junk fragments, it also splices in a recursive update (i.e., using the &&-operator) with the current annotation. Figure 6 shows the pattern that results from applying this default pre-processing to the pattern specified in Figure 5. Of course, the default can be overridden by specifying the full pattern.

The definition of current annotations, and their use in the junk fragments, reflects the role loop invariants play in the Hoare-calculus. Since the loop invariant contains all information required to prove the body, all irrelevant loops (i.e., barriers) in the body need to maintain it, and all relevant loops (i.e., nested loops) need to contain a complete invariant as well as a sufficient post-condition by themselves.

4.4 Pattern Refinement

The primary declarative knowledge used to certify a class of auto-generated programs with respect to a safety property is the set of annotation schemas. Although schemas provide a concise and

```
(for (I := _ to _)@ IndexI do {
   (* ∌ for J := _ to _ do {
         (* ∌ (A[(I ; J) ⇜ (J ; I)] := _ ∌ A)) ;
         (A[(I ; J) ⇜ (J ; I)] := _ ∌ A)
         *
      }) ← &&(inv InvI);
   (for (J := _ to _)@ IndexJ do {
      (* ∌ (A[(I ; J) ⇜ (J ; I)] := _ ∌ A)) ← &&(inv InvI ∧ InvJ);
      ((A[(I ; J) ⇜ (J ; I)])@ AI := _ ∌ A) ← &(post PostAI)
      * ← &&(inv InvI ∧ InvJ ∧ PostAI)
   }) ← &(inv InvJ, post PostJ)
   * ← &&(inv InvI ∧ PostJ)
}) ← &(inv InvI, post PostI)
```

Figure 6. Pre-processed version of the `for_for_assign` pattern

```
schema(for_assign_refined
, _
, def(A)
, (for I := _ to _ do A[_ ; I] := _ ∌ A) ∉ (for _ do _)
, default
, []
, []
).
```

Figure 7. Refined annotation schema `for_assign_refined`

high-level representation of this knowledge, it is clearly desirable to minimize the number of schemas. One approach is to make the schemas sufficiently general so that they can handle all cases of interest (e.g., all properties, as in Figure 4). However, not only will it never be possible for the author of a schema to anticipate all the cases it might be applied to, it is often not the most elegant solution to cram excessive functionality into one schema.

Instead, we allow a simple form of schema reuse, based on pattern refinement. The idea is that a schema should be applicable to nodes representing code fragments which have been matched against a pattern that is more refined than the schema's pattern.

Suppose for an example that we want to use one schema for two different code generators, but have additional information for one generator about the occurrences of this pattern. Consider for example the schema `for_assign` shown in Figure 4, which allows arbitrary dimensional arrays. Now assume we know that the generator only generates matrix-based code (i.e., the array will be two-dimensional), and further, that it represents vectors as matrices of dimension $1 \times N$ (i.e., the loop index variable will only occur in the second position), which allows us to to simplify the pattern. Assume further that we need to ensure that this pattern is not applied within double-nested loops initializing "proper" matrices, requiring an outside-constraint. Rather than copying and modifying the `for_assign`-schema, we simply define a new, skeletal schema `for_assign_refined` (see Figure 7) that uses the refined pattern, but contains no actions. This allows us to further simplify the pattern by removing all weaving actions and access operators from the pattern. We then use `for_assign_refined` only for CFG-construction (where it results in the desired matches), and rely on the unchanged `for_assign` to construct the annotations.

Conceptually, this requires the patterns to have the same "computational content", that is, to differ only in the occurring constrained patterns and meta-variable instantiations. This can be formalized as an inductive relation over patterns, but is implemented simply by checking that the matched term in the CFG matches the more refined pattern.

4.5 Schema Compiler

Since we are building on an existing, large infrastructure code base, the actual annotation schema compiler is surprisingly small—approximately 500 lines of Prolog code. It provides two top-level functions, corresponding to the phases (i.e., CFG construction and traversal) of our analysis. Both functions take as input a list of annotation schemas, but not necessarily the same. The first function simply pre-processes the patterns, strips away all `@`- and `←`-operators, and passes the result into the CFG-construction. The second function is the compiler proper. For each schema, it produces a corresponding `annotate` clause that is called from the existing inference algorithm when it is trying to annotate a CFG-node (Section 3.3). Each clause consists of five general phases: (i) check that the program fragment corresponding to the CFG-node matches the schema's pre-processed pattern (this is necessary because the two phases can use different schemas); (ii) select the program fragment and bind the pattern's meta-variables, including those introduced by pre-processing; (ii) evaluate the schema's guards, to to ensure applicability; (iv) execute the schema's actions, to construct the annotations; and finally (v) execute the update actions specified in the pattern. This is the same structure as the manually implemented annotation clauses, which is hardly surprising, since both are called in the same context. However, the schemas are significantly more compact and on average amount to only about 35% of the manual versions, and the annotation schema compiler eliminates the tedious term-operations in steps (ii) and (v) above, which are also a source of errors that are difficult to trace.

5. Experiences

We have evaluated the schema compiler and inference engine on code generated by two in-house code generators, AUTOFILTER and AUTOBAYES, as well as a COTS generator, Real-Time Workshop.

We give a more detailed evaluation for our own generators using two series of models, `orb` and `segm`, for AUTOFILTER and AUTOBAYES, respectively. Both generate code that is highly numerical, using many vector and matrix operations, and with complex control flow.

`orb` is an idealized model of the orbital dynamics of the Crew Exploration Vehicle using a simple aiding sensor for position and velocity.[6] It assumes that the earth is a perfect ellipse and is formulated as a two-body problem using Kepler's Laws [27]. AUTOFILTER generates Kalman filter based state estimation code from this, which estimates the state of the CEV from the sensor readings. `orbj2` extends `orb` by adding so-called J2 perturbations. These are additional terms in the differential equations of the process model of the vehicle dynamics which account for irregularities in the earth's gravitational field. $\texttt{orbj2}_{\texttt{bier}}$ represents the same model but where the generator is configured to select a different algorithm, namely the Bierman measurement update. This uses LU matrix decomposition in order to represent matrices in a more numerically stable form.

$\texttt{segm}_{\{1,2,3\}}$ are three different program versions generated by AUTOBAYES from the same model, by using different initialization methods for an iterative clustering algorithm. These programs have been applied to an image segmentation problem for planetary nebula images taken by the Hubble Space Telescope.

[6] This model was developed by the first author together with Johann Schumann, and is based on a model of the orbital coasting mode of the space shuttle developed by the second author.

Spec.	$\|P\|$	$\|A\|$	N	VC	T_{inf}	T_{VCG}	T_{simp}	T_{ATP}
orb	326	398	2	22	2.2	0.1	3.2	24
orbj2	378	424	2	22	2.7	0.1	3.7	25
$\texttt{orbj2}_{\text{bier}}$	447	2106	3	53	5.2	0.1	5.2	71
$\texttt{segm}_1$	165	1521	3	105	4.3	0.0	4.8	88
$\texttt{segm}_2$	161	1495	2	107	4.6	0.0	5.0	86
$\texttt{segm}_3$	155	1512	2	107	4.5	0.0	4.8	90

Table 1. Annotation inference: results for *init*-property

Spec.	$\|P\|$	$\|A\|$	N	VC	T_{inf}	T_{VCG}	T_{simp}	T_{ATP}
orb	326	78	0	0	0.1	0.1	1.5	-
orbj2	378	96	0	0	0.2	0.1	1.7	-
$\texttt{orbj2}_{\text{bier}}$	447	208	0	7	0.2	0.1	2.2	4.4
$\texttt{segm}_1$	165	125	0	0	0.1	0.0	0.3	-
$\texttt{segm}_2$	161	129	1	4	0.3	0.0	0.4	3.1
$\texttt{segm}_3$	155	148	1	4	0.3	0.0	0.4	3.2

Table 2. Annotation inference: results for *array*-property

5.1 Initialization Safety

Table 1 shows the results of applying the inference engine for the *init*-safety property to the code generated from the above models by AUTOFILTER and AUTOBAYES.

The first two columns give the size of the generated programs and the size of the inferred annotations, which are as large as, and in some cases substantially larger than, the program itself. The third column gives the number of definition patterns used to generate the annotations for each program. This is, in contrast, quite small—in each case here, only either 2 or 3 patterns are required to handle the programs. This is partly because the junk mechanism allows a single high-level pattern to capture much of the variability present in the code, and confirms our intuition that our pattern language is a highly concise means of encapsulating the knowledge required to prove safety properties.

The next column gives the number of verification conditions generated from the annotated program. The additional algorithmic complexity for $\texttt{orbj2}_{\text{bier}}$ is reflected in a substantially larger number of VCs, although it requires only one more pattern.

The subsequent columns lists the times taken to infer the annotations, to apply the VCG, to simplify and to prove the VCs. Inference time is clearly negligible in comparison to prover time, which dominates the overall run-time. All times here are wall-clock times in seconds, measured on an otherwise idle 2.2GHz standard PC with 3GB RAM running Red Hat Enterprise Linux WS release 4. We used the SSCPA system [26] to run the E (version 0.99) [25] and SPASS (version 2.2) [29] theorem provers in parallel.

5.2 Array Safety

Table 2 shows the results of applying the inference engine for the *array* safety property to the same models and generator configurations. This property is significantly simpler than *init*, and this is reflected in both the number of definition patterns, and the number of VCs. In fact, for most of the cases here, there are no definitions required. This is a consequence of no uses being designated hot [8]. There are, nevertheless, still some annotations generated (simple loop bounds which do not require patterns). In several cases, the VCs are simplified away entirely before the prover phase.

The only cases which require definition patterns are $\texttt{segm}_2$ and $\texttt{segm}_3$, which make use of array indirection, and so require annotations to give bounds on the values of matrix elements.

Spec.	$\|P\|$	$\|A\|$	N	VC	T_{inf}	T_{VCG}	T_{simp}	T_{ATP}
orb	326	230	2	2	1.1	0.0	53.1	38
orbj2	378	256	2	2	1.3	0.0	54.5	38

Table 3. Annotation inference: results for *symm*-property

5.3 Matrix Symmetry

Some of the matrices handled by the Kalman-filter algorithm represent covariance information [16] and should thus maintain their symmetry after each update within the Kalman filter loop. This can be formulated as a domain-specific safety property. In the corresponding annotation schemas, the patterns describe the different update steps for the covariance matrices. Formulating the schemas was straightforward, and yielded the correct annotations at the first attempt. However, this reused insights and code from our previous work on generating annotations together with the code (see [7] for details). Schema formulation "from scratch" would require some initial domain engineering. For example, the Bierman measurement update requires additional schemas; consequently, Table 3 contains no entries for $\texttt{orbj2}_{\text{bier}}$.

The inferences time are longer than for the simpler properties but still remain small. Since there are only a few updates to the covariance matrices, *symm* only induces a small number of VCs. However, these are substantially more complicated, which is reflected in larger simplification and proof times. Moreover, neither E nor SPASS were able to cope with the large number of matrix axioms in the domain theory, so we switched to using E-Setheo [22] as prover here.

5.4 Optimizing Generators

Since we consider the code generator as a black box, and make no assumptions about the way the code is generated, but only rely on its final form, our approach is also applicable to optimizing code generators. The key step is to formulate patterns and schemas that appropriately capture the result of applying the optimizations rather than encoding them in our framework.

Fortunately, the existing patterns are—in combination with the default pattern pre-processing—insensitive to many commonly applied optimizations, including common subexpression elimination, loop hoisting, and loop fusion. Consider for example an unoptimized fragment on the left and assume this is optimized as shown on the right:

```
for i := 1 to N do                for i := 1 to N do
  for j := 1 to M do                v := 1/i*i;
    a[i,j] := 1/i*i;                for j := 1 to M do
for i := 1 to N do                    a[i,j] := v;
  for j := 1 to M do                  b[i,j] := v+1;
    b[i,j] := a[i,j]+1;
```

In both cases, the `for_for_assign` schema is applicable. The reason for this insensitivity is the list wildcard patterns added during pre-processing. These absorb the code fragments introduced or moved into a new location by the optimizations. In the unoptimized case, each pair of loops will become a definition node for the respective initialized variable (with the other pair becoming a barrier node), and the list wildcards will be set to empty. In the optimized case, the fused loops will become the definition for both variables, and the list wildcards will be matched against the assignments to `v` and the other array-variable. Note that this causes the program fragment (i.e., the fused loop) to be annotated multiple times (with different annotations), but this is also possible for unoptimized code.

Other optimizations can change the program structure beyond what can be handled by the list wildcards alone. In these cases

```
schema(for_assign_seq_binary
, init
, def(A)
, (for (I := _ to _)@ Index do
     (A[(_ ⊅ I)@LoL; I ⋘ I;(_ ⊅ I)@LoR]) := _ ⊅ A
      ...
     A[(_⊅I)@HiL; I ⋘ I;(_⊅I)@HiR]) := _ ⊅ A) ← &(post PostAI)
  ) ← &(inv Inv, post Post)
, default
, [nonvar(LoL), nonvar(HiL) ;
   nonvar(LoR), nonvar(HiR) ]
, [anngen_for(init, AI,
              Index, _ := LoL to HiL,
              Inv, Post, _, PostAI)]).
```

Figure 8. Annotation schema `for_assign_seq_binary`

it is necessary to develop new annotation schemas. Consider for example loop unrolling: assume that $M = 3$, and that the inner loop is expanded to give:

```
for i := 1 to N do {
  v := 1/i*i;
  a[i,1] := v;
  b[i,1] := v+1;
  a[i,2] := v;
  b[i,2] := v+1;
  a[i,3] := v;
  b[i,3] := v+1;
}
```

At first glance, it looks like the `for_assign` schema (see Figure 4) would now be applicable here. However, the constraints on the added list wildcards (see Figure 6) rule it out, which is the "right thing" to do: with a more permissive pattern, `for_assign` would pick up only the first of the unrolled assignments, and would consequently construct an incomplete (but correct) annotation, which would lead to unprovable proof obligations.

A dedicated schema for this class of fragments is shown in Figure 8. The main difference to the simple `for_assign` schema is that the pattern in the loop's body now explicitly checks for the unrolled sequence of assignments, again relying on the preprocessing to match away any junk. Since either of the two loops could have been unrolled, the pattern allows the remaining index variable to occur in either position, and the schema's guards ensure that it occurs consistently. Note that the missing index is simply constructed "on-the-fly" from the lower and upper array indices, using a fresh Prolog variable as index variable

5.5 Real-Time Workshop

In a separate project, we have begun to apply the AUTOCERT technology to Real-Time Workshop (RTW) [1]. RTW is a commercial off-the-shelf system that generates C code from Simulink models. This is ongoing work at an early stage; since the integration infrastructure is not complete, we need to simplify and manually translate the RTW output into the internal representation used by AUTOCERT. However, our initial experience is encouraging.

Here, we concentrate on *init*-safety. We have analyzed a small program corpus from an independent NASA project and found the expected regularity in the code structure. Moreover, within this corpus, most idioms were already covered by patterns originally developed for AUTOFILTER and AUTOBAYES, and we only had to formulate two new patterns. The first of these patterns handles a straightforward combination of the idioms shown in Figures 1(a) and (b) (i.e., a **for**-loop followed by a sequence of assignments), while the second is more interesting. It handles situations like

```
for i := 1 to N do a[i] := b[i];
for i := 1 to N do a[i+N] := c[i];
```

where different arrays are copied into into different segments of the same target array.

6. Related Work

Annotation inference, or invariant generation, is an active research area. Approaches use both static and dynamic program analysis methods, and can further be distinguished according to the category of the inferred annotations: we can contrast type annotations, where properties are checked by special type systems, with logical annotations, which are usually processed by a VCG and then a general-purpose theorem prover. Our work is in the latter category.

Early approaches [10, 18, 28] are based on predicate propagation and 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 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.

Kovács and Jebelean [19] use techniques from algebraic combinatorics and polynomial algebra to compute polynomial relations between variables that are assigned to within loops. These relations are then turned into annotations and supplied to a VCG. The aim is to characterize the behavior of loop variables in order to prove the functional correctness of numeric procedures. They are able to precisely characterize the class of loops for which they can infer annotations, although users must manually add any non-algebraic assertions (e.g., inequalities) which are required. Abstract interpretation has also been used to infer annotations in separation logic for pointer programs [20] although the techniques required there are fairly specialized and elaborate compared to our patterns.

AOP is usually concerned with dynamic properties of programs but [21] gives a language, inspired by description logic, for describing static properties of programs. Their pattern language has some similarities to ours, but is used to define pointcuts that match against violations of design rules, and the advice is simply the corresponding error message. Since they are concerned with localizing errors, there is no need to infer annotations or propagate information throughout the program. Our pattern language also captures static properties but, in contrast, is essentially used to match against fragments which establish the specified property.

Chin et al. [3] observe that the need for type annotations can also be a significant burden for the use of type-based safety frameworks. They give a type system for an OO kernel language which enforces safety properties via annotations. The properties are defined for objects given as ADTs (e.g., sparse array, bounded buffer) supplied with appropriate annotations (size invariants and pre-conditions), which must, however, be written manually.

Deputy [5] is a dependent type system for C which can be instantiated by type constructors used to express safety properties of mutable data structures. Programs are annotated with programmer supplied type annotations, which are extended with automatically inferred derived information. The type inference is implemented as a program transformation which inserts run-time checks that the properties expressed by the annotated types are preserved. A subsequent analysis phase can omit some of these checks.

In generate-and-test methods 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 [11] is a dynamic annotation inference tool. 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 [24] with the ESC/Java static checker [15]. However, like all dynamic annotation generation techniques, it 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. Houdini [14] is a static generate-and-test tool that uses ESC/Java to statically refute invalid candidates. Houdini starts with a candidate set for the entire program and then iterates 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

We have presented a declarative annotation schema language and a schema compiler which, together with a generic annotation inference engine, forms the AUTOCERT system, which is able to automatically certify a wide range of auto-generated programs with respect to various safety properties—in fact, almost the entire range of models and configurations (i.e., algorithmic variants and optimizations) for AUTOBAYES and AUTOFILTER. This continues the work begun in [8] and represents a significant advance in both power and expressivity of the technique.

By raising the level of abstraction at which annotation knowledge is expressed, we are able to concisely capture many variations of the underlying code idioms. In particular, we can easily deal with optimizations which obscure low-level code structure. In this vein, we plan to extend the framework with *pattern isomorphisms*. This will complement the current use of the simple theory matching and will lead to simpler patterns and less schemas. Indeed, there are other forms of guidance which are naturally expressed in a similarly declarative fashion, and we view schemas as a first step towards a fully programmable certification language.

We are currently extending AUTOCERT with more domain-specific policies. We are also scaling up the system to the interprocedural level. This requires not only rather straightforward extensions in the VCG, but also interface specifications to record safety assumptions about procedure parameters. We expect inference times to increase, but there are numerous optimizations in the core inference engine that we expect will keep the approach practically viable. Finally, we continue to work on an adaptation to the Real-Time Workshop [1] code generator. Initial results are encouraging.

References

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

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

[3] W.-N. Chin et al. Verifying Safety Policies with Size Properties and Alias Controls. In *ICSE'05*, pp. 186–195. ACM Press, 2005.

[4] M. Clavel et al. The Maude system. In *RTA-10*, *LNCS 1631*, pp. 240–243. Springer, 1999.

[5] J. Condit et al. Dependent Types for Low-Level Programming. In *ESOP'07*, pp. 520–535. Springer, 2007.

[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 and B. Fischer. A generic annotation inference algorithm for the safety certification of automatically generated code. In *GPCE'06*, pp. 121–130. ACM Press, 2006.

[9] 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.

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

[11] 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.

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

[13] B. Fischer and E. Visser. Retrofitting the AutoBayes program synthesis system with concrete syntax. In C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors, *Domain-Specifi c Program Generation*, *LNCS 3016*, pages 239–253. Springer, 2004.

[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. S. Grewal and A. P. Andrews. *Kalman Filtering: Theory and Practice Using MATLAB*. Wiley Interscience, 2001. 2nd edition.

[17] 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.

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

[19] L. Kovács and T. Jebelean. Finding Polynomial Invariants for Imperative Loops in the Theorema System. In *Proc. IJCAR'06 Workshop Verify'06*, pp. 52–67, 2006.

[20] 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.

[21] C. Morgan, K. De Volder, and E. Wohlstadter. A Static Aspect Language for Checking Design Rules. In *AOSD '07*, pp. 63–72. ACM Press, 2007.

[22] M. Moser et al. The Model Elimination Provers SETHEO and E-SETHEO, *J. Automated Reasoning*, **18**(1997) 237–246.

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

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

[25] S. Schulz. E – A Brainiac Theorem Prover. *J. AI Communications*, 15(2/3):111–126, 2002.

[26] G. Sutcliffe and D. Seyfang. Smart selective competition parallelism ATP. In *FLAIRS'99*, pp. 341–345. AAAI Press, 1999.

[27] D. A. Vallado. *Fundamentals of Astrodynamics and Applications*. Space Technology Library. Microcosm Press and Kluwer Academic Publishers, second edition, 2001.

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

[29] C. Weidenbach et al. SPASS Version 2.0. In *Proc. 18th CADE*, *LNAI 2392*, pp. 275–279. Springer, 2002.

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

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