Generating Customized Verifiers for Automatically Generated Code

Ewen Denney

Research Institute for Advanced Computer Science
NASA Ames Research Center
M/S 269-2, Moffett Field, CA 94035, USA
Ewen.W.Denney@nasa.gov

Bernd Fischer

School of Electronics and Computer Science
University of Southampton
Southampton, SO17 1BJ, England
B.Fischer@ecs.soton.ac.uk

Abstract

Program verification using Hoare-style techniques requires many logical annotations. We have previously developed a generic annotation inference algorithm that weaves in all annotations required to certify safety properties for automatically generated code. It uses patterns to capture generator- and property-specific code idioms and property-specific meta-program fragments to construct the annotations. The algorithm is customized by specifying the code patterns and integrating them with the meta-program fragments for annotation construction. However, this is difficult since it involves tedious and error-prone low-level term manipulations.

Here, we describe an approach that automates this customization task using generative techniques. It uses a small *annotation schema compiler* that takes a collection of high-level declarative *annotation schemas* tailored towards a specific code generator and safety property, and generates all customized analysis functions and 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 required. The improvements described here make it easier and faster to customize the system to a new safety property or a new generator, and we demonstrate this by customizing it to certify frame safety of space flight navigation code that was automatically generated from Simulink models by MathWorks' Real-Time Workshop.

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

General Terms Algorithms, Verification

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

GPCE'08, October 19–23, 2008, Nashville, Tennessee, USA.

1. Introduction

The verification of program safety and correctness using Hoare-style techniques requires many 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 their construction difficult and expensive. For example, proving 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.

However, in certain cases it is possible to construct the required annotations automatically, e.g., if the program comes from a limited domain [16] or if only limited properties are shown [14]. In our previous work [10], we have developed a generic *annotation inference algorithm* that exploits the idiomatic structure of automatically generated code to weave in the annotations required to verify a given safety property. Idioms are recurring code patterns that solve similar programming tasks using similar constructions. In automatically generated code, they result from the way generators usually derive code, i.e., by combining a finite number of building blocks (e.g., templates) following a finite number of combination methods (e.g., template expansion). For example, Figure 1 shows three matrix initialization idioms employed by Real-Time Workshop; the code in Figure 1(c) uses a vector to represent the matrix.

```
A[0,0 ]:= a_{0,0};          for i:= 0 to n do          for i:= 0 to n -1 do
...                           for j:= 0 to m do          for j:= 0 to m do
A[0,m]:= a_{0,m};             A[i,j]:= a;                A[i* n +j]:= a;
A[1,0 ]:= a_{1,0};
...
A[n,m]:= a_{n,m};
```

(a) (b) (c)

Figure 1. Idiomatic matrix initializations in Real-Time Workshop

Our inference algorithm uses generator- and property-specific patterns to capture these code idioms and property-specific meta-program fragments associated with these patterns to construct the annotations. It first builds an abstracted control-flow graph (CFG), using the patterns to collapse the code idioms into single nodes. It then traverses this graph and follows all paths from use-nodes backwards to all corresponding definitions, adding the annotations along the way. This algorithm is implemented as part of our AUTOCERT system for the safety certification of automatically generated code. Its core (i.e., CFG construction and transversal) is fully generic but it must be 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.

Here, we describe an approach that largely automates this customization task. It uses a small *annotation schema compiler* that 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 generating a customized annotation inference algorithm. The compiler allows us to represent all 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. It 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. Together with improvements of the underlying core inference algorithm and the pattern matching machine also described here, the schema compiler makes it much easier and faster to customize the generic annotation inference algorithm to a new safety property or a new generator. We demonstrate this by customizing it to certify frame safety of space flight navigation code that was automatically generated from Simulink models by Real-Time Workshop [1].

In this paper, we thus build on but substantially improve over our previous work on annotation inference for automatically generated code [10]. Our paper makes four main technical contributions. The first two are (i) the development of the schema compiler and (ii) the implicit junk handling by the compiler. In addition, we have also (iii) modified the underlying core inference algorithm so that the inference for one variable can "trigger" the inference for other variables if the safety of the former depends on the latter. This dependency is also controlled by the schemas. Finally, we have (iv) extended the pattern language by additional constraint operators, which make it more expressive and allow more context-sensitivity in the patterns, thus minimizing reliance on the use of arbitrary meta-programming functionality in the guards. In particular, we have integrated a simple data-flow analysis into the matcher, which allows us to match a pattern against the *content* of a variable as well. This significantly improves our ability to distinguish structurally equivalent code fragments. Our main empirical contribution here is a significantly extended evaluation of our general annotation inference approach. In particular, we have evaluated AUTOCERT using C code generated by the Real-Time Workshop code generator. Based on the extensions described here, we have been able to certify frame and initialization safety for code generated from Simulink and Embedded Matlab models, as well as several safety properties for a variety of programs generated by our AUTOBAYES [13] and AUTOFILTER [27] generators.

The next section gives some general background on the safety certification of automatically generated code and summarizes the underlying annotation inference algorithm as far as is required here; more details can be found in our previous work [10]. Section 3 explains the extended pattern language used here. 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. Technical Background

Here, we briefly summarize our approach to safety certification of automatically generated code and the generic annotation inference algorithm. Details can be found in our previous work [8, 9, 10].

2.1 Safety Certification

Program Safety Safety certification demonstrates that a program does not violate certain conditions during its execution. A *safety property* [8] is an exact semantic characterization of these conditions, while a *safety policy* is a set of specialized Hoare rules designed to show that a program satisfies the safety property of interest. Language-specific properties can be applied to all programs in the underlying programming language. For example, variable initialization before use (*init*) ensures that each variable or individual array element has been explicitly assigned a value before it is used, while array bounds safety (*array*), requires each access to an array element to be within the specified upper and lower bounds of the array. Our approach can also be used with more specific domain-specific properties. For example, frame safety (*frame*) shows that vehicle navigation software uses the different frames of reference consistently [18, 23].

Annotation and Verification We split certification into an untrusted annotation construction phase (see below for details) 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. As usual in Hoare-style verification, a VCG traverses the 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 serve as "hints" or lemmas for the ATP, and must be established in their own right. Consequently, they remain untrusted—a wrong annotation cannot compromise the assurance provided by the system.

2.2 Idioms

The idioms used by a code generator are essential to our approach because they (rather than the generator's building blocks or combination methods) determine the interface between the generator and the inference algorithm. The idioms and corresponding patterns are specific to the given safety property, but the inference algorithm remains the same for each property. This allows us to apply our technique to black-box generators as well, as the example of Real-Time Workshop shows. Moreover, it also allows us to handle optimizations: as long as the resulting code remains idiomatic, neither the specific optimizations nor their order matter. We can thus customize a verifier for a given generator and safety property, by identifying the relevant idioms and formalizing them as patterns.

The idioms represent the key knowledge that drives the annotation inference. However, we need to distinguish different classes of idioms, in particular, definitions, uses, and barriers. *Definitions* 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 appear between definitions and uses (in the control flow graph) that require annotations, i.e., principally loops. In the case of initialization and frame 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 all cases, barriers are loops.

2.3 Inference Algorithm Structure

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, which corresponds to the safety condition which must hold at that point in the code.

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" and "hot uses" described in our previous work [10]. Intuitively, a variable use (and thus the variable) is hot, if there is a barrier between the use location and any of the variable's definitions. 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 CFG and iterates over all paths in the CFG that start with a hot use, before it finally constructs the annotations for the paths.

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 below for details). The CFGs are abstracted by collapsing entire code idioms matching specific patterns into individual nodes. Since the patterns can be parametrized 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 by Harrold and Rothermel [15].[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 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.

Annotation of Paths For each hot use of a hot variable, the path computation 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 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.

[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.

$$
\begin{array}{lll}
P ::= & x & x \in X \\
 & \mid f(P_1, \ldots, P_n) & f \in \Sigma \\
 & \mid \_ \mid P\texttt{?} \mid P\texttt{*} \mid P\texttt{+} \mid P_1 \ldots P_2 & \\
 & \mid P_1 \texttt{ ; } P_2 \mid P_1 \texttt{ || } P_2 \mid P_1 \texttt{ <+ } P_2 & \\
 & \mid P_1 \texttt{ // } P_2 \mid P_1 \texttt{ \textbackslash\textbackslash } P_2 & \textit{(lookahead)} \\
 & \mid P_1 \supset P_2 \mid P_1 \not\supset P_2 \mid P_1 \notin P_2 & \textit{(subterm matches)} \\
 & \mid P \leftarrow U & \textit{(weave)} \\
 & \mid P \texttt{ @ } x & \textit{(access)} \\
 & \mid P \texttt{ :: } C & \textit{(constraint)} \\
U ::= & \texttt{\&(}A\,\{\texttt{,}\,A\}\texttt{)} \mid \texttt{\&\&(}A\,\{\texttt{,}\,A\}\texttt{)} & \\
 & \mid \langle \textit{prim-op} \rangle & \\
A ::= & \textbf{inv}\ F \mid \textbf{pre}\ F \mid \textbf{post}\ F & F \in \mathcal{F} \\
C ::= & P_1 \texttt{ ~= } P_2 & \textit{(data-flow lookback)} \\
 & \mid \langle \textit{prim-op} \rangle &
\end{array}
$$

Figure 2. Grammar of extended pattern language

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 (which are typically loops), and basic nodes which represent loops that have not been matched by any other pattern. However, the most important (and interesting) class is the definitions because their annotations (more precisely, their final post-conditions) are used as initial values for annotation along the paths.

For example, we can define a separate annotation schema for each of the three different initialization blocks shown in Figure 1. Each schema inserts a final (outer) post-condition establishing that the matrix x is initialized, e.g., in the first two cases $\forall\, 0 \leq i \leq N, 0 \leq j \leq M \cdot \mathsf{A}_{\text{init}}[i,j] = \text{INIT}$.

However, the annotations 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, the annotation schema itself might still fail. For example, the pattern in the schema handling the idiom in Figure 1(a) simply matches against a sequence of assignments, but the schema requires that the indices of the first and last assignments are the lower and upper bound of the array, respectively. Of course, even if the schema succeeds, the generated VCs might fail since annotation construction is untrusted. In other words, matching is approximate, but ultimately checked by the prover.

3. 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 pattern language is essentially a tree-based regular expression language similar to XML-based languages like XPath [3]; Figure 2 shows its grammar. Compared to our previous work [10], we added more contextual patterns (the lookahead operators `//` and `\\` and the outside-operator $\notin$), operators to support interactions with the meta-program fragments constructing the actual annotations, and constraints (`::`), in particular the data-flow lookback operator `~=`.

Core Patterns The language supports matching of tree literals $f(P_1, \ldots P_n)$ over a given signature Σ, wildcards (`_`) and the usu-

al regular operators for optional (**?**), list (*****) and non-empty list (**+**) patterns, as well as alternation (||) and concatenation (;) operators. The ellipsis operator ... 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.

Context Dependencies 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, contextual patterns and pattern meta-variables. Contextual patterns generalize the idea of lookahead that is well-known from regular expression matching. A *contextual 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 context pattern P_2 that can rule out potential base matches, depending on the given context operator *op*. Possible operators are lookahead (//) and its complement (i.e., P_1 \\ P_2 matches if P_1 is not followed by P_2), which check the right siblings of the term matched against the base pattern (i.e., work horizontally), and various forms of subterm matching, which 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 least one subterm that matches P_2; similarly, $P_1 \not\ni P_2$ matches all terms that match P_1 and have no subterm that matches P_2.[2] For example, the pattern A**[* ∌** I **; I ; * ∌** I**] := _ ∌** A uses subterm matching to rule out array updates in which the index variable I appears more than once in the index list, or in which the array A appears on the right-hand side of the assignment. In contrast to the inward-looking operators ⊃ and ∌, the ∉-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`.

Interaction Operators Another extension of the pattern language describes interactions with the meta-program fragments constructing the actual annotations. The two operators ↼ and @ are used to compile the guards and actions of the corresponding schema. The weave-operation $P \leftharpoonup U$ 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 meta-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.

Constraints Constraints are similar to contextual patterns in the sense that the base pattern P will be returned as result only if the constraint C is satisfied. C is either a data-flow lookback (see below), or an arbitrary meta-program operation. These can for example be used to check structural properties of the match that cannot be expressed in the pattern language, e.g., identical lengths of two different lists.

Data-flow Lookback Since pattern matching works on the syntactic structure of the program, all relevant semantic differences must be reflected syntactically. However, in practice, this is often not the case and semantically different concepts are represented by syntactically similar code. For example, in vehicle navigation software, frames of reference are used to represent different coordinate systems within which the position and orientation of objects are measured.[3] Transformations between different frames can be represented by a direction cosine matrix (DCM) [23]; Figure 3 shows the different structure of two example DCMs transforming from the NED frame into two different target frames.

$$\begin{pmatrix} -\cos\lambda\sin\phi & -\sin\lambda & -\cos\lambda\cos\phi \\ -\sin\lambda\sin\phi & \cos\lambda & -\sin\lambda\cos\phi \\ \cos\phi & 0 & -\sin\phi \end{pmatrix} \quad \begin{pmatrix} \cos(H-A) & \sin(H-A) & 0 \\ -\sin(H-A) & \cos(H-A) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 3. DCM matrices: (a) *NED-to-ECEF* (b) *NED-to-Nav*

For the certification *frame* safety (i.e., all measurements are transformed into the right frames before they are processed) we need to be able to distinguish between the two different DCMs, but the code generated by Real-Time Workshop uses temporary variables to store the elements, and the matrix (represented as a vector) is updated using these (see Figure 4(a)). Note that additional temporaries are used to factor out common subexpressions. In order to identify the sequence of array updates as the *DCM-NED-to-ECEF* idiom, and to distinguish it from the structurally equivalent *DCM-NED-to-Nav* idiom, we thus need to match the *content* of the variables `v0` to `v8` (and thus the content of the meta-variables x_0 to x_8) against the respective patterns.

```
c0:=-1
...
w0:=cos(in5)
w1:=sin(in4)
w2:=sin(in5)
...
v0:=c0*w0*w1;
v1:=c0*w1*w2;
...
v8:=c0*w1;
...
a[0]:=v0;
a[1]:=v1;
...
a[8]:=v8;
```

(a)

$(A[0] := x_0) :: (x_0 \tilde{=} -cos(L) * sin(P));$
$(A[1] := x_1) :: (x_1 \tilde{=} -sin(L) * sin(P));$
...
$(A[8] := x_8) :: (x_8 \tilde{=} -sin(P))$

(b)

Figure 4. *DCM-NED-to-ECEF* code fragment (a) and pattern (b)

Rather than using arbitrary meta-programs to analyze the program structure, we introduce a specific constraint operator that triggers a simple, approximate data-flow analysis to infer possible symbolic values of program variables that are then checked against the constraint pattern. Figure 4(b) shows the actual pattern used to capture the idiom. The schema itself is shown in Section 5.2.

[2] In our previous work [10], these were denoted by $P_2 \in P_1$ and $P_2 \notin P_1$, respectively.

[3] Here we consider the vehicle-centered systems North-East-Down (NED) and wander azimuth (Nav), and the earth-centered systems Earth-Centered Inertial (ECI) and Earth-Centered Earth Fixed (ECEF).

The structural core of the pattern is simply the sequence of array updates, but each right-hand side is constrained by an appropriate lookback. When an update such as `a[0]:=v0` is matched, the meta-variables A and x_0 are instantiated with `a` and `v0`, respectively, and then the data-flow lookback constraint on the instantiation of x_0 is checked. The data-flow analysis thus looks back through the program to find possible values for `v0`. The preceding assignment yields `c0*w0*w1` for which a match is attempted against the constraining pattern $-cos(L) * sin(P)$. This attempt fails, which triggers further lookbacks to values of the variables occurring in the value found for the original variable `v0`, i.e., `c0`, `w0`, and `w1`. Using the theory matching described below, the lookback eventually succeeds, with the meta-variables L and P instantiated with `in5` and `in4`, respectively. Note that a "plain" lookback (i.e., a reverse lookahead) would remain insufficient in such situations, since the required value of x_0 is only constructed in several steps and several different locations.

The dataflow lookback is only an approximation, since it ignores control flow predicates and CFG back edges. However, this approximation remains safe, as all matches are checked by the VCs and thus ultimately by the ATP.

Match procedure 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. Constraints are checked whenever a match for the base pattern has been found. However, the dataflow lookback requires interaction with the CFG construction and the term traversal, as traversed terms need to be pushed on a stack for later inspection.

The match procedure also supports a limited form of matching modulo theory: users can specify how tree literal patterns can be mapped onto terms. We use this to handle some irrelevant syntactic variance in the programs, for example, to handle commutative operators such as addition and multiplication by simply checking all possible permutations of the operands, or 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; in the worst case, the theory implementation may not terminate, which will of course cause the non-termination of the entire matching procedure.

4. Annotation Schema Representation and Compilation

An *annotation schema* is a declarative representation of all knowledge required to handle a class of specific certification situations. A schema includes 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. In practice, however, guards are rarely required, and none of the schemas shown here uses them.

The AUTOCERT 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 AUTOCERT's core annotation inference algorithm outlined above and described in more detail in our previous work [10], which is implemented in Prolog, the output of the compiler is simply a set of Prolog clauses.

```
schema(for_assign
, SP
, def(A)
, (for (I := _ to _)@ Index do
     ((A[*⊅ I; I; *⊅ I])@ AI := _ ⊅ A) ← &(post SC)
  ) ← &(inv Inv, post Post)
, default
, []
, [safe(SP, AI, SC),
   ind_schema(step1, SC, [Index], [Inv,Post])]
) :- SP=init ; SP=range(_).
```

Figure 5. Annotation schema `for_assign`

4.1 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 the 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 `for_assign` shown in Figure 5,[4] which comes from Simulink, we can thus use the same schema (with appropriately parametrized 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 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** $\forall\ 1 \leq\ j < i\ \cdot\ a_{\text{init}}$**[**$j$**]** $=$ INIT **do**
  a**[**i**]** **:=** b**[**i**];**
  post a_{init}**[**i**]** $=$ INIT
post $\forall\ 1 \leq\ j \leq\ N\ \cdot\ a_{\text{init}}$**[**$j$**]** $=$ INIT

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

[4] 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.

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 the post-condition on the individual array-update, which will be used to prove the loop's post-condition. This yields

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

Since this schema requires no guards, the final step is to add the actions that actually construct the annotations. Here, the actions consist of calls to the safety predicate `safe` and the annotation construction predicate `ind_schema` (see Section 4.2) to construct the post-condition for a single array-update and the loop invariant and post-condition, respectively. The predicates require access to specific parts of the actual program fragment matched against the pattern, e.g., 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 5). In the above example, we thus get the annotations $SC \equiv a_{\text{init}}[i] = \text{INIT}$, $Inv \equiv \forall\, 1 \le j < i \cdot a_{\text{init}}[j] = \text{INIT}$, and $Post \equiv \forall\, 1 \le j \le N \cdot a_{\text{init}}[j] = \text{INIT}$, as expected.

4.2 Induction Schemas

Schemas can make use of arbitrary specialized meta-programming in order to construct annotations but, in general, most annotations encapsulate general induction principles, so we use a generic predicate `ind_schema` to construct them. This takes the form of induction to use, the base formula (usually the safety predicate on the hot variable), and the indices (i.e., bound variables and bounds) to induct over, and returns the list of annotations.

Several types of induction are currently supported. The schemas discussed here use single- (`step1`) and doubly-nested induction (`step2`), which constructs the necessary inner and outer invariants and post-conditions. There is also a schema that handles diagonal matrix traversals `diag`.

We keep the induction schemas separate from the annotation schemas themselves for two main reasons. First, the induction schemas encapsulate general induction principles that work for multiple annotation schemas so that very few of them are needed. Second, an annotation schema does more than an induction schema. The latter just constructs some annotations, but the former says where to put those annotations, how to pre-process the pattern, under what conditions (i.e., guards) they should apply, and whether there are any dependent variables (see Section 4.4).

4.3 Pattern Pre-processing

Often, even auto-generated code does not exactly fit the pattern specified in a schema, but contains "junk", i.e., additional 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_lin` schema shown in Figure 6, which annotates two nested **for**-loops initializing a single matrix A that is represented as a vector; here, the constraint $N + 1 \equiv N'$ symbolically evaluates whether the multiplier N' has the right value. This schema should also apply in situations where the outer loop contains additional statements before or after the inner 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.

```
schema(for_for_assign_lin
, init
, def(A)
, (for (I := 0 to N)@ IndexI do
     (for (J := 0 to _)@ IndexJ do
        (((A[I*N'+J])@ AIJ : : N+1≡N') := _ ⊅ A) ← &(post SC)
     ) ← &(inv InvJ, post PostJ)
  ) ← &(inv InvI, post PostI)
, default
, []
, [safe(init, AIJ, SC),
   ind_schema(step2, SC, [IndexI, IndexJ],
              [InvI, PostI, InvJ, PostJ])]
).
```

Figure 6. Annotation schema `for_for_assign_lin`

Extending the schema to cover these cases requires two steps. 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 wildcards (see Figure 7). 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 start annotating 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 7 shows the pattern that results from applying this default pre-processing to the pattern specified in Figure 6. 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.

```
(for (I := 0 to N)@ IndexI do {
   (* ∄ for J := 0 to _ do {
         (* ∄ (((A[I*N'+J])@ AIJ : : N+1≡N') := _ ∄ A)) ;
          (((A[I*N'+J])@ AIJ : : N+1≡N') := _ ∄ A)
         *
       }) ← &&(inv InvI);
   (for (J := 0 to _)@ IndexJ do {
      (* ∄ (((A[I*N'+J])@ AIJ : : N+1≡N') := _ ∄ A)
      ) ← &&(inv InvI ∧ InvJ);
       (((A[I*N'+J])@ AIJ : : N+1≡N') := _ ∄ A) ← &(post SC)
      * ← &&(inv InvI ∧ InvJ ∧ SC)
   }) ← &(inv InvJ, post PostJ)
   * ← &&(inv InvI ∧ PostJ)
}) ← &(inv InvI, post PostI)
```

Figure 7. Pre-processed version of the pattern used in the `for_for_assign_lin` schema

```
schema(mtrans_int
, frame
, def(A)
, (C := 0;
    ((for (I := 0 to N)@ IndexI do
       (for (J := 0 to M)@ IndexJ do {
          (((A[I+N'*J])@ AIJ : : N+1≡N') := T[C];
          C++
       }) ← &(inv C=J+N'*I ∧ FPre ∧ InvJ,
              post C=M+1+N'*I ∧ FPre ∧PostJ)
     ) ← &(inv C=N'*I ∧ FPre ∧ InvI, post  FPre ∧ PostI)
    ) ← &(post FPre ∧ A=trans(T))
   ) ← &(pre FPre, post FPost)
, default
, [T]
, []
, [FPre  = has_frame(T, dcm(F2, F1)),
   FPost = has_frame(A, dcm(F1, F2)),
   ind_schema(step2, AIJ=T[J+N'*I],
              [IndexI, IndexJ],
              [InvI, PostI, InvJ, PostJ])]
).
```

Figure 8. Annotation schema `mtrans_int`

4.4 Dependent Hot Variables

The inference first passes over the program to determine the hot variables before it proceeds along every path from every hot use until either a definition or the beginning of the program is reached. Sometimes, however, a definition will trigger further hot variables that could not be (efficiently) detected on the first pass. This happens, intuitively, when one variable is computed from another. For example, in the schema `mtrans_int` shown in Figure 8, the variable A is computed as the transpose of T, so that its frame depends on T's frame.[5] The `mtrans_int` schema uses a syntactic variant, where the additional (sixth) argument `[T]` indicates that T is a dependent hot variable for A. Inference will thus proceed past this definition for A and restart, looking for a definition for the new hot variable T. Specifying the dependent hot variables is straightforward using the schemas, which shows the power of the approach. In the previous system version using manual annotation clauses, computing the dependent hot variables could require the implementation of complex term decomposition.

[5] Note that the schema has three nested post-conditions: the first (i.e., on the outer loop) states the element-wise definition of the transpose; the second "lifts" this to an explicit transpose operator; and the third uses this to derive the appropriate frame information.

4.5 Schema Compiler

Since we are building on AUTOCERT's existing, large infrastructure code base, the actual annotation schema compiler is surprisingly small—approximately 1000 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. This allows us for example to use a schema with a more refined pattern to construct the CFG, but to re-use a more general schema to actually construct the annotations. The first function simply preprocesses the patterns and uses the pre-processed patterns for 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 2.3). Each clause consists of six 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; (iii) evaluate the schema's guards, to ensure applicability; (iv) execute the schema's actions, to construct the annotations; (v) execute the update actions specified in the pattern; and finally, (vi) processes the dependent hot variables, if any are specified. In addition, the compiler also generates several auxiliary functions required by the inference algorithm, e.g., extracting the overall post-condition attached to a 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 schema compiler eliminates the tedious term-operations in steps (ii) and (v) above, which are a source of errors that are difficult to trace and clutter up the manually implemented annotation clauses. Consequently, the schemas are significantly more compact and on average amount to only about 35% of the manual versions.

Since we took care to generate code that is compatible with the existing code base, only minor modifications were required to the rest of AUTOCERT in order to interface it with the schema compiler. The other extensions described here, in particular the data-flow lookback and the dependent hot variables, required more substantial changes to the system. However, these extensions were designed to handle a wider range of certification problems and are orthogonal to the schema compiler itself.

The annotation schemas could also be interpreted at inference time, rather than being compiled upfront. However, the use of Prolog as AUTOCERT's implementation language means that we could simply compile the schemas on-the-fly, and thus achieve the same effect as with an interpreter. Moreover, an interpretation would require more substantial modifications to the existing AUTOCERT implementation, and minimizing such modifications was the main motivation for choosing the compilation approach.

5. Evaluation

We have evaluated the schema compiler and its interaction with AUTOCERT's core inference engine on code generated by two in-house code generators, AUTOFILTER and AUTOBAYES, as well as a COTS generator, Real-Time Workshop, which generates code with distinct characteristics from several modeling languages. Here we look at code generated from Simulink and Embedded Matlab models.

5.1 AUTOBAYES and AUTOFILTER

We originally developed the annotation schema compiler for use with our AUTOBAYES and AUTOFILTER generators. Both generate numerical code that uses many vector and matrix operations, and has complex control flow with nested loops, but they work in different domains: AUTOBAYES generates statistical data analysis code, while AUTOFILTER is tailored towards state estimation problems. Tables 1 and 2 summarize the evaluation of both systems.

For AUTOBAYES, we use three different program versions $\texttt{segm}_{\{1,2,3\}}$ generated 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. They have been used in our previous work on annotation inference [10], which allows us to compare the results of the annotation schema compiler with manually implemented annotation code. For this application, certifying different safety properties is not required; however, it increases our confidence in the overall correctness of the AUTOBAYES system.

For AUTOFILTER, we used a series of idealized models of the orbital dynamics of the Crew Exploration Vehicle using a simple aiding sensor for position and velocity.[6] `orb` assumes that the earth is a perfect ellipse and is formulated as a two-body problem using Kepler's Laws [23]. 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. Generating code for these models required extension to AUTOFILTER, which rendered obsolete the manually implemented annotation clauses used in our previous work.

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, both in non-blank lines of code. Note that the annotations are as large as, and in some cases substantially larger than, the program itself. The third column gives the number of schemas used to generate the annotations for each program. This is, in contrast, quite small—in each case here, only either 2 or 3 schemas 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 the schema language is a highly concise means of encapsulating the knowledge required to prove safety properties. In total, we needed only 8 and 6 schemas for each of AUTOBAYES and AUTOFILTER, respectively, to formalize initialization safety; 5 of these are shared between both systems. Translating the existing manually implemented annotation clauses into new schemas was straightforward. Adapting the system to the new `orb`- and `orbj2`-code required only a few iterations to get the annotations right and the VCs proven. In our experience, this adaptation process has now become much simpler and faster than it had been in the old approach using the manually implemented annotation clauses.

[6] These models were developed by the first author together with Johann Schumann, and are based on a model of the orbital coasting mode of the Space Shuttle developed by the second author.

Spec.	$\lvert P\rvert$	$\lvert A\rvert$	N	VC	T_{inf}	T_{VCG}	T_{ATP}
$\texttt{segm}_1$	182	1521	3	105	4.3	4.8	88
$\texttt{segm}_2$	178	1495	2	107	4.6	5.0	86
$\texttt{segm}_3$	172	1512	2	107	4.5	4.8	90
`orb`	326	398	2	22	2.2	3.3	24
`orbj2`	378	424	2	22	2.7	3.8	25
$\texttt{orbj2}_{\texttt{bier}}$	447	2106	3	53	5.2	5.3	71

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

Spec.	$\lvert P\rvert$	$\lvert A\rvert$	N	VC	T_{inf}	T_{VCG}	T_{ATP}
$\texttt{segm}_1$	182	125	0	0	0.1	0.3	-
$\texttt{segm}_2$	178	129	1	4	0.3	0.4	3.1
$\texttt{segm}_3$	172	148	1	4	0.3	0.4	3.2
`orb`	326	78	0	0	0.1	1.6	-
`orbj2`	378	96	0	0	0.2	1.8	-
$\texttt{orbj2}_{\texttt{bier}}$	447	208	0	7	0.2	2.3	4.4

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

The next column gives the number of verification conditions generated from the annotated program. The additional algorithmic complexity for $\texttt{orbj2}_{\texttt{bier}}$ is reflected in a substantially larger number of VCs, although it requires only one more pattern. The subsequent columns list the times taken to infer the annotations, to apply the VCG (which includes simplification) and to prove the VCs. Inference time is clearly negligible in comparison to prover time, which dominates the overall run-time.[7] Since we trust the Hoare-rules of the safety policy, the axioms of the domain theory, and the theorem provers, the fact that all VCs are proven indirectly validates our schemas.

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 [10]. There are, however, 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. Each example requires a single schema, which was again straightforward to formulate.

5.2 Real-Time Workshop: Simulink

We used AUTOCERT to generate a customized verifier for showing frame safety of C code generated from Simulink models by Real-Time Workshop. We then used this verifier on a navigation subsystem currently under commercial development for NASA, which transforms the coordinate frames of various signals. The signals represent state information using quaternions and the software converts the quaternions to and from direction cosine matrices (DCMs), so that matrix algebra can be used to perform the transformation. Several DCMs (NED-to-Nav, NED-to-ECEF, and ECI-to-ECEF are constructed directly using standard trigonometric formu-

[7] 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 [22] to run the E (version 0.999) [21] and SPASS (version 3.0c) [25] theorem provers in parallel.

```
schema(dcm_ned_ecef
, frame
, def(A)
, ((A[0]:=x0) : : (x0 ~= -cos(L) * sin(P));
   (A[1]:=x1) : : (x1 ~= -sin(L) * sin(P));
   (A[2]:=x2) : : (x2 ~=  cos(P));
   (A[3]:=x3) : : (x3 ~= -sin(L));
   (A[4]:=x4) : : (x4 ~=  cos(L));
   (A[5]:=x5) : : (x5 ~= 0);
   (A[6]:=x6) : : (x6 ~= -cos(L) * cos(P));
   (A[7]:=x7) : : (x7 ~= -sin(L) * cos(P));
   (A[8]:=x8) : : (x8 ~= -sin(P))
   ) ← &(post has_frame(A, dcm(ned, ecef)),
         pre ∃ λ, φ · has_unit(λ, geolong) ∧ has_unit(φ, geolat)
             ∧ x0=-cos λ sin φ ∧ x1=-sin λ sin φ
             ∧ x2=cos φ ∧ x3=-sin λ ∧ x4=cos λ ∧ x5=0
             ∧ x6=-cos λ cos φ∧x7=-sin λ cos φ∧x8=-sin φ)
, none
, []
, []
).
```

Figure 9. Annotation schema `dcm_ned_ecef`

las and taking various physical quantities either as input from the signals or as constants, namely, geodetic latitude, longitude, time, true heading, platform azimuth, and the Earth's rotational velocity [23].

`nav3` and `nav5` represent two different conceptual components of the navigation subsystem that carry out specific transformations. $\texttt{nav5}_2$ is generated from a model equivalent to `nav5`, but using different Real-Time Workshop configuration settings; consequently, the generated code is quite different. There are numerous other subsystems not discussed here that use the same basic components. `nav3` and `nav5` were chosen to minimize functional overlap, so that they actually comprise most of the blocks in the subsystem. In all cases, AUTOCERT was provided with assumptions on the frames and physical units of the input signals, and the aim of the verification was to establish that the output, a quaternion state vector, is in the correct coordinate frame. Table 3 shows the results. Note that the size for `nav3` and `nav5` includes *both* components, since Real-Time Workshop merges them into a single program.

`nav5` and $\texttt{nav5}_2$ use the schema `dcm_ned_ecef` shown in Figure 9, whereas `nav3` uses a similar schema `dcm_ned_nav` (not shown here, but see Figure 3 for the structure of the required DCM). In total, *frame* safety requires 15 schemas. Of these, 7 describe specific transformations like `dcm_ned_ecef`, which could be transcribed directly from the literature. The remaining schemas formalize the effects of the applied matrix operations, including some of Matlab's built-in functions. These schemas represent a substantial domain analysis effort in a mathematically challenging domain—it took approximately one month to analyze the given code base, to understand the domain concepts and their implementation, and to formulate the patterns and the required annotations. However, about 90% of the effort was related to the domain analysis and would have been required for a one-off safety proof of the code base as well; only the remaining 10% was directly related to the schema formulation.

The proof times shown in Table 3 are substantially longer than for *init* and *array*, reflecting the more complex mathematical reasoning that is required. As before, the inference time is negligible in comparison to the proof times. The proofs of these VCs also require a logical theory of matrix and frame algebra but this is orthogonal to the development of the schemas, and is not discussed here. However, the development of this logical theory in a form that was suitable for the automated provers was actually the most labor-intensive aspect of the certification.

Spec.	$\lvert P\rvert$	$\lvert A\rvert$	N	*VC*	T_{inf}	T_{VCG}	T_{ATP}
`nav3`	807	383	6	33	2.2	20.6	350
`nav5`	807	307	6	31	2.2	10.6	315
$\texttt{nav5}_2$	309	298	6	27	1.6	7.6	289

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

5.3 Real-Time Workshop: Embedded Matlab

Embedded Matlab is a mathematical scripting language which allows the use of functions and equations in models. Variables in the equations typically represent vectors and matrices and therefore the generated code is heavily loop-based, and quite different in character from code generated from "pure" Simulink.

Here we illustrate the *init* certification of code generated from an Embedded Matlab model consisting of four matrix equations from a Kalman filter. The generated code is about 150 LOC and could be certified using just two schemas, of which one could even be reused from AUTOBAYES/AUTOFILTER. The other schema needed, `for_for_assign_lin` (cf. Figures 1(c) and 6), is specific to Embedded Matlab.

```
for i0 := 0 to N
inv ∀ 0≤i<i0, 0≤j≤N · x_init[i + j * 2] = INIT do
  for i1 := 0 to N
  inv ∀ 0≤i, j ≤N·
        (i < i0 ∨ (i = i0 ∧ j < i1)) ⇒ x_init[i+j*2] = INIT do
    x11:= 0;
    for i2 := 0 to N
    inv ∀ 0≤i, j ≤N · x11_init = INIT ∧
          (i<i0 ∨ (i=i0 ∧ j < i1))⇒ x_init[i + j * 2]=INIT do
      x11 +:= bv0[i2+i1*2] * dv0[i0+i2*2];
    x[i0+i1*2] := x11+R[i0+i1*2];
  post ∀ 0≤i≤i0, 0≤j≤N · x_init[i + j * 2] = INIT
post ∀ 0≤i, j≤i0 · x_init[i + j * 2] = INIT
x11 := x[0];
d := x[1]*x[2] - x11*x[3];
x[0] := x[3] / d;
x[3] := x11 / d;
x[1] := -x[1] / d;
x[2] := -x[2] / d;
post ∀ 0≤i≤3 · x_init[i] = INIT
```

Figure 10. Annotated Embedded Matlab code

In Figure 10, we show the fragment which uses the two schemas, including the annotations generated by the schemas; we omitted constraints on the loop variables to simplify the presentation. The array `x`, which represents a 2x2-matrix, is first assigned via a doubly-nested for-loop and then inverted via a sequence of assignments. Since inference works backwards through the CFG, the assignment sequence is annotated first. By setting `x` as its own dependent variable, inference can then proceed on to the loop.

5.4 Optimizing Generators

One of the advantages of the annotation schemas is their ability to specify patterns at a high-level and let the machinery handle the variability in the code. 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, therefore, to optimizing code generators. In particular, the

existing patterns are—in combination with the default pattern preprocessing—insensitive to many commonly applied optimizations, including common subexpression elimination, loop hoisting, and loop fusion.

We have exploited this to handle optimizations in the Real-Time Workshop generators for Simulink and Embedded Matlab. Consider for example the unoptimized fragment on the left, which is optimized (using loop hoisting and loop fusion) 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 (which is a generalization of the `for_assign` schema to nested loops) is applicable. The reason that `for_for_assign` remains insensitive to the optimization is the list wildcard patterns added during preprocessing. 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. In the case of Embedded Matlab, the `for_for_assign_lin` schema is also able to absorb the effects of these optimizations in the same way.

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. However, these approaches generally hard-code specific domain knowledge and cannot be customized simply, if at all, in the same way our approach allows.

Early approaches [11, 24] are based on predicate propagation and use inference rules similar to a strongest post-condition 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 [16] 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 [17] although the techniques required there are fairly specialized and elaborate compared to our patterns.

Generate-and-test approaches use a fixed pattern catalogue to construct candidate annotations and then try to validate (or refute) them, using static or dynamic methods. 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. Daikon [12] 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 with ESC/Java [20]. However, like all dynamic annotation generation techniques, it remains incomplete because it relies on a test suite to generate the candidates and can thus miss annotations on paths that are not executed often enough.

The specific problem of frame safety has been addressed by Lowry et al. [18], who used a domain-specific type system to verify the safety of abstract geometric calculations. The language analyzed was quite simple, however, so that annotations could be restricted to the declarations of the input variables, with no need for the inference of patterns or intermediate annotations. Although the underlying domain knowledge is similar to what we use for the frame example, this is a very specific solution, in contrast to our "retargetable verifier".

AOP is usually concerned with dynamic properties of programs but Morgan et al. [19] give 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.

Antkiewicz et al. [4] use code queries, which are approximations to structural and behavioral patterns, in order to reverse engineer framework-specific models from framework code. It is similar to our work in the sense that we use patterns to reverse engineer "logical structure".

Conventional static analysis tools based on abstract interpretation, such as PolySpace [2], are notoriously inaccurate. Although improvements have been made (e.g., with Astree [7]), such tools can only handle relatively simple safety properties, and are unable to produce the detailed explanations AUTOCERT provides in the form of proofs, safety cases [5], and safety documents (work in progress).

Coccinelle [6] uses model checking over the CFG to identify source code fragments that need modification in response to a patch. The underlying logic CTL-VW allows both universally and existentially quantified variables (compared to the existential interpretation of the meta-variables in our pattern language), but it is restricted to control flow only, and does not take any data flow into account.

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. We have developed a set of schemas which customizes AUTOCERT for certifying the frame safety of navigation code generated from Simulink models. Other sets of schemas support the certification of code generat-

ed from Embedded Matlab, as well as the entire range of models and configurations (i.e., algorithmic variants and optimizations) for AUTOBAYES and AUTOFILTER. The underlying inference technique is independent of the generator, but relies on the idiomatic structure of the generated code. For the examples reported in this paper, we were able to identify the necessary code idioms; however, more work is necessary to determine how well the technique works for other properties and other generators.

This paper continues previous work [10] 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.

Our system currently comprises approximately 50 schemas for the *array*, *init*, and *frame* safety properties. We are developing additional sets of schemas and extending the schema language itself to support the certification of other properties and of code generated from a wider range of models. There are various physical and geometric properties that can be analyzed similarly to coordinate frames, such as the correct use of Euler angles, quaternion handedness, and so on, and we plan to adapt the frame schemas for those properties. Currently, the inference is restricted to an intraprocedural analysis, although it can handle calls to annotated library procedures. This is sufficient for the generators we have used so far, but we are planning to extend the system towards an interprocedural inference.

Finally, although our emphasis so far has been on certifying safety, the schema language and inference engine are not limited to this and, in fact, several of the schemas we have presented here are actually verifying full functional correctness of certain fragments in order to establish safety. For example, in order to verify frame safety for the examples above, we need to verify the correctness of the underlying matrix transformations. Similarly, the various DCM schemas are effectively functional verifications of those constructions. We intend to further explore the possibilities for functional verification. Likewise, there is no need to restrict AUTOCERT tool to automatically generated code—it can just as well be applied to manually written code, with the proviso that the less idiomatic the code is, the less accurate the analysis will be, and we intend to explore this avenue as well.

Acknowledgments This paper is based on work supported by NASA under award NCC2-1426. and NNA07BB97C. Fischer is supported by the EPSRC grant EP/F052669/1.

References

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

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

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

[4] M. Antkiewicz, T. Tonelli Bartolomei, and K. Czarnecki. Automatic extraction of framework-specific models from framework-based application code. In *ASE'07*, pp. 214–223. ACM, 2007.

[5] N. Basir, E. Denney, and B. Fischer. Constructing a Safety Case for Automatically Generated Code from Formal Program Verification Information. In *SAFECOMP'08, LNCS 5219*, pp. 249–262. Springer, 2008.

[6] J. Brunel et al. A Foundation for Flow-Based Program Matching Using Temporal Logic and Model Checking. Technical Report 08/2/INFO, Ecole des Mines de Nantes, 2008.

[7] P. Cousot et al. The Astreé analyzer. In *ESOP'05, LNCS 3444*, pp. 21–30. Springer, 2005.

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

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

[10] 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.

[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] 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.

[16] 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.

[17] 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.

[18] M. Lowry, T. Pressburger, and G. Roşu. Certifying domain-specific policies. In *ASE'01*, pp. 118–125. IEEE, 2001.

[19] 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.

[20] 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.

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

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

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

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

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

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

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