Annotation Inference for Safety Certification of Automatically Generated Code
(Extended Abstract)

Ewen Denney
USRA/RIACS, NASA Ames
edenney@email.arc.nasa.gov

Bernd Fischer
ECS, University of Southampton
B.Fischer@ecs.soton.ac.uk

1 Introduction

Automated code generation is an enabling technology for model-based software development and promises many benefits, including higher quality and reduced turn-around times. However, the key to realizing these benefits is generator correctness: nothing is gained from replacing manual coding errors with automatic coding errors.

Since the direct verification of generators is unfeasible with existing techniques, "correct-by-construction" approaches have been explored. However, these remain difficult to implement and to scale up, and have not seen widespread use. Currently, generators are validated primarily by testing [8], though this cannot guarantee correctness and quickly becomes excessive. Here we follow an alternative approach based on the observation that the correctness of the generator is irrelevant if instead the correctness of the generated programs is shown individually. Similar to proof carrying code [7], we focus on the Hoare-style certification of specific safety properties. This simplifies our task but still leaves the problem of constructing the appropriate logical annotations (i.e., pre-/postconditions and loop invariants), due to their central role in Hoare-style techniques.

In the *certifiable program generation* approach, the code generator itself is extended in such a way that it generates the necessary annotations together with the code [3]. This has two major disadvantages. First, the developers need to modify the code generator in order to integrate the annotation generation but sources are often not accessible, in particular for commercial generators. Second, it is difficult to implement and to maintain because the annotations are cross-cutting concerns, both on the object-level (i.e., the generated program) and the meta-level (i.e., the generator).

Here we describe an alternative technique that uses a generic post-generation annotation inference algorithm to circumvent these problems. We exploit both the highly idiomatic structure of automatically generated code and the restriction to specific safety properties. Since generated code only constitutes a limited subset of all possible programs, the new "eureka" insights required in general remain rare in our case. Since safety properties are simpler than full functional correctness, the required annotations are also simpler and more regular. We can thus use patterns to describe all code constructs that require annotations and templates to describe the required annotations. We use techniques similar to aspect-oriented programming to add the annotations to the generated code: the patterns correspond to (static) point-cut descriptors, while the introduced annotations correspond to advice.

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

Figure 1. Idiomatic matrix initializations

The annotation inference algorithm can run completely separately from the generator and is generic with respect to the safety property, although we use initialization safety as running example here. It has been implemented and applied to certify initialization safety for code generated by AutoBayes [5] and AutoFilter [10].

2 Background

Idiomatic Code. Automated code generators derive low-level code from declarative specifications. Approaches vary, but for our purposes the details do not matter, and we build on a template-based approach. What does matter, however, is the fact that most generators produce *idiomatic code* (i.e., code that exhibits a regular structure beyond the syntax of the programming language) by combining a finite number of building blocks. For example, AutoBayes and AutoFilter only use three templates to initialize a matrix, resulting in either straight-line code or one of two doubly-nested loop versions (cf. Fig. 1).

The idioms are essential to our approach because they (rather than the templates) determine the interface between the code generator and the inference algorithm. They can be recognized from a given code base alone, even without knowing the templates that produced the code. This allows us to apply our technique to black-box generators as well.

Safety Certification. The purpose of safety certification is to demonstrate that a program does not violate certain conditions during its execution. A *safety property* is an exact characterization of these conditions. A *safety policy* is a set of Hoare rules designed to show that safe programs satisfy the safety property of interest. In our framework the rules are formalized using the usual Hoare triples extended with a "shadow" environment which records safety information related to the corresponding program variables, and a *safety predicate* that is added to the computed verification conditions (VCs) [2].

We use initialization safety as example but other safety properties can also be used with our algorithm, including absence of out-of-bounds array accesses and nil-pointer dereferences [2, 7]. Initialization safety ensures that each variable or individual array element has been explicitly assigned a value before it is used. The shadow variable $\bar{x}$ contains the value INIT after the variable x has been assigned a value; shadow arrays capture the status of the individual elements. Only rules for statements assigning a value to a location affect the shadow environment; the most complicated cases are the loop rules which require explicit invariants. Initialization safety defines an expression to be safe if the respective shadow variables have the value INIT, so that, e.g., *safe*$(x\,[\,i\,])$ simply translates to $\bar{\imath} = \text{INIT} \wedge \bar{x}\,[\,i\,] = \text{INIT}$.

VC Processing and Annotations. A VC generator (VCG) traverses the annotated code and applies the Hoare rules to produce VCs, starting with the postcondition *true*. If all VCs are proven by an automated theorem prover (ATP), the program is safe wrt. the safety property. However, the ATP has no access to the program internals; hence, all pertinent information must be taken from the annotations which, in general, must be so detailed that their inference is intractable. For safety certification, however, the Hoare rules are specialized and the safety predicates are regular and simple, so that the required annotations are simpler.

3 Inference Algorithm

Our aim is to "get information from definitions to uses", i.e., to annotate the program such that the VCG has the information necessary to show the program safe (wrt. the given property) as it works its way back through the program. The notions of definitions and uses are specific to the given safety property. For initialization safety, definitions correspond to the different initialization blocks as shown in Fig. 1, while uses are statements which read a variable. For array bounds safety, definitions are the array declarations since the shadow variables get their values from the declared bounds, while uses are statements which access an array variable.

Fig. 2(a) shows an example program, similar to code produced by AUTOFILTER, that initializes two vectors `A` and `B` and computes the sums `s` and `t` of their respective elements. AUTOFILTER's target language is a simple imperative language with basic control constructs (i.e., **if** and **for**) and numeric scalars and arrays as the only datatypes. It also supports domain-specific operations like matrix assignment and multiplication that are not used in the example.

Top-level Algorithm Structure. The inference algorithm first scans the code for relevant variables. For each variable, it builds an abstracted control flow graph where irrelevant parts of the program are collapsed into single nodes. It then follows all paths backwards from the variable's use nodes until it encounters either a cycle or a definition node for the variable. Paths that do not end in a definition are discarded and the remaining paths are traversed node by node. Annotations are added to all intermediate nodes that otherwise constitute barriers to the information flow before the definitions themselves are annotated.

Patterns and Pattern Matching. We use patterns to capture the idiomatic code structures and pattern matching to find the corresponding code locations. Our pattern language is a tree-based regular expression language similar to XPath. It supports matching of tree literals, wildcards (_), optional (?), list (*) and non-empty list (+) patterns, as well as alternation (||) and concatenation (;). $P_1 \in P_2$ matches all terms that match P_2 and have at last one subterm that matches P_1 and similarly for $P_1 \notin P_2$. We use meta-variables in patterns to introduce context dependency: an uninstantiated meta-variable matches any term but it then becomes instantiated and subsequently matches only other instances of the matched term. 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`. Meta-variables are instantiated eagerly but instantiations are undone if the enclosing pattern fails later on.

Hot Variable Identification. Proving a program safe requires annotations at the points where the VCG needs information about the contents of the essential shadow variables. The algorithm thus first passes through the program to determine which variable uses are essential or "hot", i.e., for which there are barriers (mainly loops) to the information flow along the paths to all definitions. Since the system constructs annotations only for these hot variables, they must be approximated conservatively; we focus on the hot uses of `A` and `B` in lines 5.2 and 5.3.

Abstracted Control Flow Graphs. The algorithm uses abstracted control flow graphs (CFGs), in which code fragments matching specific patterns are collapsed into individual nodes. Since the patterns can depend on the variables, separate abstracted CFGs must be constructed for

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

Figure 2. (a) Original program (b) Abstraction for `B` (c) Annotations for `B` (d) Abstraction for `A` (using *block*- and *barrier*-patterns) (e) Annotations inferred for `A` and `B`

each hot variable. CFG construction 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 pattern, rather than recursively descending into the statements. The algorithm constructs *use*- and *def*-nodes and uses *barrier*- and *block*-nodes as optimizations to represent code that can be regarded as opaque (to different degrees) because it contains no definition for the given variable. Both are treated as atomic nodes during path search, which drastically reduces the number of paths that need be explored. *barrier*-nodes represent code that requires annotations, mainly loops. They must be re-expanded and traversed during the annotation phase of the algorithm. *block*-nodes are irrelevant to the hot variable because they neither require annotations (i.e., contain no barriers) nor contribute to annotations (i.e., contain no definition). They remain atomic during the annotation phase, i.e., are not entered on path traversal. Blocks are usually loop-free sequences of assignments and conditionals.

Annotation of Nodes and Paths. For `B`, the CFG construction identifies the **for**-loop in lines 3.1-3.2 as the definition for the entire array `B` and abstracts it into the definition node *def*(`B[1:N]`), cf. (Fig. 2(b). The path search then starts at the hot use in line 5.3 (abstracted into *use*(`B`)) and goes straight back up to the **for**-loop at line 5.1, where it splits. One branch passes through the bottom of the loop body but this immediately leads to a cycle and is therefore discarded. The other branch continues through lines 4.1 and 4.2 to terminate at the definition node at line 3.1. Since all branches have been exhausted, there is only one path along which annotations need to be added. The annotation process starts with the use and proceeds towards the definition terminating the path. Every node needs to be inspected, but in this case only the **for**-loop at line 5.1 requires an invariant. Since the traversal must take control flow into account, the current annotation is computed from the WPC of the previous annotation for the current node. This can be considerably more complicated than for the simple example given, but in general the form of all annotations is fully determined by the safety property, and in the case of a definition, its known syntactic structure as described by the pattern. The annotation knowledge is represented by *annotation schemas*, which are the core of the whole system. They take a match (identifying the pattern and the location), and use meta-programming to construct and insert the annotations. Here, the definition is a loop, so it needs a loop invariant and a postcondition. Since the safety property is initialization safety, both invariant and postcondition need to formalize that the shadow variable $\bar{B}$ corresponding to the current array variable `B` records the value INIT for the entries already initialized. Fig. 2(c) shows the partially annotated program after this pass.

The next pass adds the annotations for `A`. It is initialized using a different idiom—a sequence of assignments, cf. Fig. 2(d), lines 2.1–2.n—which is again collapsed into a *def*-node. The program is collapsed further by the introduction of *barrier*- and *block*-nodes. The *barrier*-nodes must be re-expanded during the path traversal phase because they require annotations (cf. line 3.1) while the *block*-nodes remain opaque. Except for this special handling, the algorithm proceeds as before, and Fig. 2(e) shows the resulting fully annotated program.

Note that the definitions and the paths are untrusted; their correctness is established by the annotations, which are themselves untrusted and ultimately checked by the ATP.

4 Experiences

We have implemented the generic inference algorithm and used an instantiation to certify initialization safety for code generated by AUTOBAYES and AUTOFILTER. This required only a small "declarative content", i.e., only pattern definitions but no changes to the core algorithm itself.

AutoFilter. For AUTOFILTER, the definitions are given by two of the matrix initialization idioms in Fig. 1, along with the scalar and matrix assignment operations **:=** and **::=**.

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

The pattern is parametrized over the hot variable x and uses "free" meta-variables i and j that are bound to the actual index variables of the matched loop. Barriers are defined as **for**-loops without any occurrence of the hot variable. Loops *with* the hot variable are then simply treated by the normal CFG-routines, i.e., not collapsed. Finally, blocks are conditionals whose branches are irrelevant because they contain no occurrence of a barrier or the hot variable.

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

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

AutoBayes. AUTOBAYES requires additional **for**-loop patterns but does not need the **::=**-pattern since it does not generate direct matrix operations. It has two additional language constructs, **abort**, which appears in the definition pattern, and **while**-loops, which can form additional barriers. Blocks and uses are the same as for AUTOFILTER.

Results. For AUTOFILTER, annotation inference proves to be very similar to the previous certifiable program generation approach. The inferred annotations are slightly larger (by 15–25%) than the generated ones but, due to simplifications, they produce fewer VCs, and all VCs are proven by the ATP. For AUTOBAYES, annotation generation has not kept up with ongoing generator development and the original annotations are now insufficient to prove the programs safe. Using annotation inference with the patterns described above, we can already certify some programs but more code patterns are required to cover the entire range of programs.

Since it needs to build and traverse the CFGs, the inference approach is slower than the generation approach, which only needs to expand templates. However, the overall proof times are comparable, indicating that the inference does not introduce new complexity for the ATP.

5 Conclusions

Early work used a combination of logical inference and heuristics to push an initial annotation forward through the program [9]; this proved to be computationally expensive and ineffective. Dynamic [4] and static [6] generate-and-test methods have also been used but they are much less goal-oriented than our approach and require refutation of many invalid annotation candidates.

In our previous certifiable program generation approach, extensions and modifications to the code generators had over time led to a situation of "entropic decay" where the generated annotations had not kept up with the generated code. The new certification system based on annotation inference is able to automatically certify the same programs as the original system, as well as some subsequent extensions. However, the re-construction is not yet complete, and we continue to extend the new system. These extensions require less effort than before since the patterns and annotation schemas are expressed declaratively and in one place, in contrast to the previous decentralized architecture where certification information is distributed throughout the code generator. Our new approach offers a general framework for augmenting code generators with a certification component, and we have started a project to apply it to MathWorks Real-Time Workshop [1].

References

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

[2] E. Denney and B. Fischer. Correctness of source-level safety policies. *FM 2003*, *LNCS 2805*, 894–913.

[3] E. Denney and B. Fischer. Certifiable program generation. *GPCE 2005*, *LNCS 3676*, 17–28.

[4] M. Ernst et al. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):1–25, 2001.

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

[6] C. Flanagan and R. Leino. Houdini, an annotation assistant for ESC/Java. *FM 2001*, *LNCS 2021*, 500–517.

[7] G. Necula. Proof-carrying code. *POPL-24*, 106–119, 1997.

[8] I. Stürmer et al. Overview of existing safeguarding techniques for automatically generated code. *SIGSOFT SEN*, 30(4):1–6, 2005.

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

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