# Constructing a Safety Case for Automatically Generated Code from Formal Program Verification Information

Nurlida Basir[1], Ewen Denney[2], and Bernd Fischer[1]

[1] ECS, University of Southampton, Southampton, SO17 1BJ, UK
(nb206r,b.fischer)@ecs.soton.ac.uk
[2] USRA/RIACS, NASA Ames Research Center
Mountain View, CA 94035, USA
Ewen.W.Denney@nasa.gov

**Abstract.** Formal methods can in principle provide the highest levels of assurance of code safety by providing formal proofs as explicit evidence for the assurance claims. However, the proofs are often complex and difficult to relate to the code, in particular if it has been generated automatically. They may also be based on assumptions and reasoning principles that are not justified. This causes concerns about the trustworthiness of the proofs and thus the assurance claims. Here we present an approach to systematically construct safety cases from information collected during a formal verification of the code, in particular from the construction of the logical annotations necessary for a formal, Hoare-style safety certification. Our approach combines a generic argument that is instantiated with respect to the certified safety property (i.e., safety claims) with a detailed, program-specific argument that can be derived systematically because its structure directly follows the course the annotation construction takes through the code. The resulting safety cases make explicit the formal and informal reasoning principles, and reveal the top-level assumptions and external dependencies that must be taken into account. However, the evidence still comes from the formal safety proofs. Our approach is independent of the given safety property and program, and consequently also independent of the underlying code generator. Here, we illustrate it for the AutoFilter system developed at NASA Ames.
**Keywords:** Automated code generation, formal program verification, Hoare logic, fault tree analysis, safety case, Goal Structuring Notation.

## 1 Introduction

Model-based design and automated code generation have become popular, but substantial obstacles remain to their widespread adoption in safety-critical domains: since code generators are typically not qualified, there is no guarantee that their output is safe, and consequently the generated code still needs to be fully tested and certified. Here, formal methods such as formal software safety certification [6] can be used to demonstrate safety of the generated code (i.e., that the execution of the code does not violate a specified property) by providing formal proofs as explicit evidence or *certificates* for the assurance claims. However, several problems remain. For automatically generated code it is particularly difficult to relate the proofs to the code; moreover, the proofs are the final stage of a complex process and typically contain many details. This complicates an intuitive understanding of the assurance claims provided by the proofs. Hence, it is important to make explicit which claims are actually proven, and on which assumptions and reasoning principles both the claims and the proofs rest. Moreover, the complexity of the tools used can lead to unforeseen interactions and thus causes additional concerns about the trustworthiness of the assurance claims. We thus believe that traceability between the proofs on one side and the certified program and the used tools on the other side is important to gain confidence in the formal certification process.

Here, we address these problems and present an approach currently under development to systematically derive safety cases from information collected during the formal software safety certification phase, in particular the construction of the necessary logical annotations. The purpose of these safety cases is to provide a "structured reading guide" for the program and the safety proofs that will allow users to understand the safety claims without having to understand all the technical details of the formal machinery. We use a fault tree analysis to identify possible risks to the program safety and the certification process, as well as their interaction logic, and thus to derive the structure of the safety cases. We then use a generic, multi-tiered argument [3] that is instantiated with respect to a given safety property and program. Its three tiers together constitute a single safety case that justifies the safety of the program. The upper tier simply instantiates the notion of safety and the formal definitions for the given safety property while the two lower tiers argue the safety of the program as governed by the property. The lower tiers are constructed individually to reflect the program structure. This can be done systematically because their structure directly follows the course the annotation construction takes through the program. In principle, our approach is thus independent of the given safety property and program, and consequently also independent of the underlying code generator [10].

We have developed the overall structure of the generic safety case and manually instantiated it for several examples, using only information logged during annotation construction. We expect that this process can be automated easily and that it will furthermore be straightforward to integrate with existing tools to construct safety cases such as Adelard's ASCE tool [1]. The program safety case will eventually be complemented by an additional safety case that will argue the safety of the underlying safety logic (the language semantics and the safety policy) with respect to the safety property (i.e., safety claims), as well as other components such as the theorem prover. This will clearly communicate how the safety claims, key safety requirements, and evidence for the program safety are connected. We expect that this will alleviate distrust in code generators, which remains a problem for their use in safety-critical applications.

## 2 Background

Here, we give a brief overview of automated code generation; we focus on the certifiable code generation approach, where the assurance is not implied by the trust in the generator but follows from an explicitly and independently constructed argument for the generated code.

### 2.1 Assurance for Automated Code Generation

Automated code generation [5] is a technique for automatically constructing software from (high-level) problem specifications or models. Code generators typically work by adapting and instantiating pre-defined code fragments for (parts of) the problem specification, and composing these partial solutions. They have a significant potential to eliminate manual coding errors and reduce costs and development times. Obviously, to realize any benefits from code generation, the generated code needs to be shown correct or at least safe. In correct-by-construction techniques such as deductive synthesis [23] or refinement [22] this is done by a mathematical meta-argument. However, such techniques remain difficult to implement and extend and have not found widespread application. A formal verification of the generator would provide a similar level of assurance, but remains unfeasible with the existing program verification techniques. Currently, generators are thus validated primarily by testing [24], in line with software development standards for safety-critical domains such as DO-178B [21]. However, this time-consuming and expensive process slows down generator development and application, and only few generators have been qualified.

We believe that product-oriented assurance approaches are a viable alternative to the process-oriented approaches outlined above. Here, checks are performed on each and every generated program rather than on the generator itself. Hence, assurance is not implied by the trust in the generator but follows from an explicitly constructed argument for the generated code. In our approach [8, 9, 11], we focus on safety properties, which are generally accepted as important for quality assurance and are also often used in code reviews of high-assurance software. We then use program verification techniques based on Hoare logic to formally demonstrate that the generated code satisfies the safety properties of interest. Our approach generally follows similar lines as proof carrying code [16] but it works on the source code level instead of the object code level [6]. However, both approaches exploit formal safety proofs as explicit evidence or *certificates* for the assurance claims over the untrusted code.

### 2.2 Formal Software Safety Certification

The purpose of software safety certification is to demonstrate that a program meets its high-level requirements and remains safe in the presence of known hazards. *Formal software safety certification* uses formal techniques based on program logics to show that the program does not violate certain conditions during its execution. A *safety property* is an exact characterization of these conditions, based on the operational semantics of the programming language. Each safety property thus describes a class of hazards. 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) [6]. However, here we focus on the information provided by constructing the annotations, and leave the details of constructing (i.e., applying the Hoare rules) and proving (i.e., calling the theorem prover) the VCs to the complementary system-wide safety case.

Formal software safety certification follows the same technical approach as program verification. A VC generator (VCG) traverses the code backwards and applies the Hoare rules to produce VCs, starting with any safety requirements on output variables. If all VCs are proven by an automated theorem prover (ATP), we can conclude that the program is safe wrt. the given safety property. This approach shift the trust burden from the program to the certification system: instead of having to trust an arbitrary program to be safe, users have to trust the certifier to be correct.

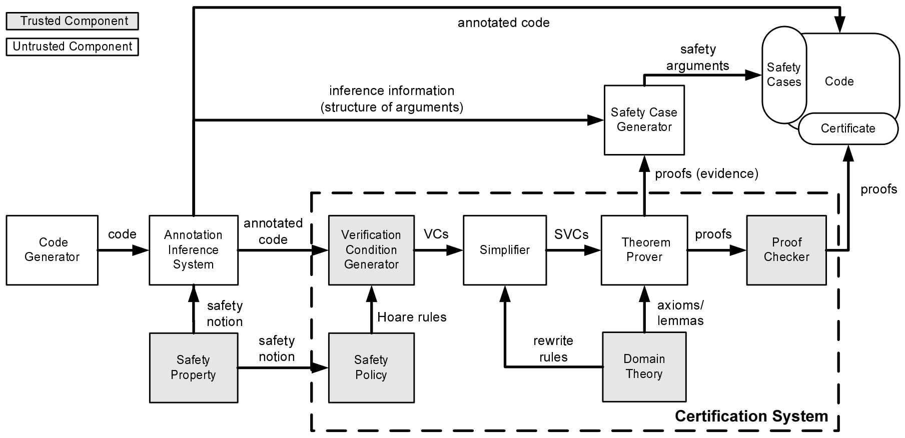

**Fig. 1.** System Architecture

Figure 1 shows the overall system architecture of our certification approach. In this, the original code generator (in this case, the AutoFilter system [28]) has been extended with the annotation inference subsystem and the standard machinery of Hoare-style verification techniques (i.e., VCG, simplifier, ATP, domain theory, and proof checker) to achieve a fully automated verification of the generated code. The architecture distinguishes between trusted (in grey) and untrusted components (in white) as shown in Figure 1. Trusted components must be correct because any errors in them can compromise the assurance provided by the overall system. Untrusted components, on the other hand, are not crucial to the assurance because their results are double-checked by at least one trusted component.These components and their interactions are described in more detail in [6, 8, 9].

Rather than acting as a black-box verification tool which provides a simple pass/fail result, our certification approach provides a structured safety arguments, supported by a body of evidence (i.e., safety cases) to demonstrate why the generated code can be assumed to be sufficiently safe. The safety case is generated from the analysis of the code and provides a high-level traceable argument of how the code complies with the specified safety property. The inference engine supplies information to the safety case generator, which renders this along with the code. The safety case generator identifies each part of the program that can draw attention to potential certification problems and select appropriate evidence to reason correctness of the underlie safety claims and the certification process. By elucidating the reasoning behind the certification process, there is less of a need to trust the tool.

Here, we use initialization safety (i.e., each variable or individual array element has explicitly been assigned a value before it is used) as an example, but our framework can handle a variety other safety properties including absence of out-of-bounds array accesses [6]; we expect that other properties handled by proof-carrying code such as null pointer dereferences [16] can be formalized easily. However, we are not restricted to showing exception freedom but can also encode domain-specific properties such as matrix symmetry or coordinate frame consistency (which requires significant proofs involving matrix algebra and functional correctness), whose violation will not immediately cause a run-time exception but still renders the code unsafe.

The Hoare-approach to safety certification is more flexible than special-purpose static analysis tools such as PolySpace [18] that can only handle the comparatively simple language-specific properties. It also provides explicit evidence in form of proofs, which static analysis tools typically lack.

## 2.3 Annotation Inference

In order to achieve a fully automated verification, a program logic requires annotations (i.e., pre- and post-conditions, and loop invariants) at key program locations. These annotations serve as lemmas that facilitate the proof of VCs, but they have to be established in their own right (i.e., they will produce VCs that show that they hold at their given location). The purpose of annotation inference [8, 9] is to construct these annotations automatically, by analyzing the program structure. In our case, the annotations must formalize all pertinent information that is necessary for the ATP to prove that all *potentially* unsafe locations are in fact safe. If the program is safe, this information will be established or "defined" at some location (which we thus call a *definition*) and maintained along all control-flow paths to all the potentially unsafe locations, where it is used. The idea of the annotation inference algorithm, therefore, is to "get the information from definitions to uses", i.e., to find the endpoints of all such generalized *def-use*-chains, to construct the formulae used in the annotations, and to annotate the program along the paths.

The annotation inference algorithm itself is generic, and parametrized with respect to a library of coding patterns that depend on the safety policy and the code generator. The use of these patterns isolates the annotation construction from the internal details of the code generation and also allows us to a certain degree to handle code that has been modified manually. The patterns characterize the notions of definitions and uses that are specific to the given safety property. For example, for initialization safety, definitions correspond to variable initializations while uses are statements which read a variable, whereas for array bounds safety, definitions are the array declarations (where the shadow variables get their values from the declared bounds), while uses are statements which access an array variable. The inferred annotations are thus highly dependent on the actual program and the properties being proven. For example, for initialization safety, an invariant on a for-loop might express that an array has been initialized up to the loop index ($\forall j \leq i \cdot A_{\text{init}}[j] = \textsc{init}$). The VCG will turn this annotation into three VCs, corresponding to establishing the invariant on loop entry, preservation of the invariant by the loop body, and implication by the “exit form” of the invariant (i.e., over the loop bounds) of the loop post-condition. For other safety properties, the annotations can be seen as encapsulating the safety requirements directly. In the case of the symmetry policy, a postcondition $\forall i, j \cdot M[i,j] = M[j,i]$ expresses the symmetry of $M$. Again, this will be converted into VCs and checked by the prover. However, it is the *def-use*-dependencies, rather than the annotations or the VCs, which govern the overall structure of both the safety argument and the safety case.

## 3 Hazard Analysis for Formal Program Verification

While formal program verification has become a viable alternative in demonstrating program safety, doubts about the trustworthiness of the verification proofs remain. These doubts concern not only the correctness of the proofs (i.e., whether each proof step is legal in the underlying calculus) or the correctness of any of the other tools that handle the verification conditions, but also the question whether the proofs actually entail program safety. Since there are many possible ways in which the trustworthiness can be compromised, a fault tree analysis is required to identify the chain of causes and their interaction logic that initiate this undesired event.

However, our situation is complicated by the fact that the code generator is a meta-level system, and we do not know the application context of the generated program. In order to analyze the situation already at this meta-level (rather than deferring this to the final application), we need to make the simplifying but conservative assumption that every violation of the safety property is a “potential condition that can cause harm to personnel, system, property or environment”, i.e., a hazard [15].

A further complication is caused by the fact that the certification system is purely observational in the sense that it cannot introduce any additional hazards as defined above, but should nonetheless be included in the hazard analysis. We thus need to look at the interaction between the code generator and the certification system to identify faults of the combined system. We consider two sets of indicators, namely the output of the code generator, or more precisely, whether the generated code is safe or unsafe, and the output of the certification system, or more precisely, its claim about the safety of the code (i.e., safe, unsafe, or unknown). We then consider all situations in which these two indicators do not agree as abnormal or faults of the combined system. The most critical fault, on which we concentrate here, occurs if the code exhibits an unsafe behavior when it is executed but the certification system claims that all safety properties were proven to hold.

The fault tree shown in Figure 2 demonstrates how the combinations of events that could lead to the top-level hazard (i.e., an undetected violation of the safety property) are linked together. It focuses on showing possible events that might invalidate the safety claim construction as it follows the structure of the generated code. A complete analysis would also need to look at other hazards, e.g., incorrect proofs or inconsistent axioms; the corresponding fault tree will lead to the system-wide safety case and is left for future work.

Figure 2 shows that there are two potential causes for the top-level hazard, either a missed potentially unsafe location in the code or the certification system erroneously concluded that all locations in the code are safe. Potentially unsafe locations in the generated code can be missed because of

- an incomplete or incorrect formalization of the safety policy corresponding to the given safety property (i.e., the failure to detect a location as potentially unsafe),
- an incomplete or incorrect representation of the safety requirements in critical annotations (e.g., a wrong global post-condition on the output variables),
- missing VCs (e.g., due to errors in the VCG), or
- incomplete coverage of the program, missing claims for any variable, occurrence or path in the program.

Here, our safety case will focus on the last cause, as it is the only cause directly related to the code generator. All other causes will be handled by the complementary system-wide safety case.

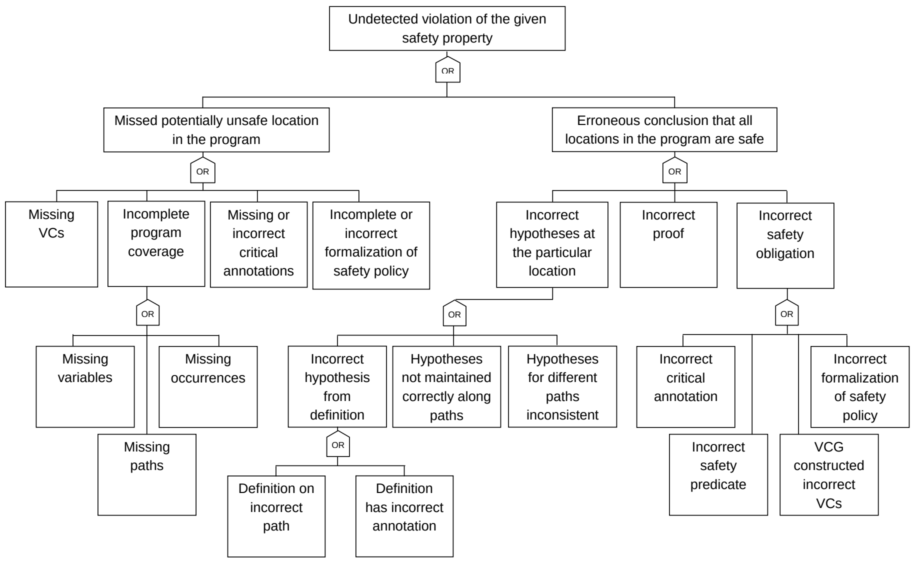
Undetected violation of the given safety property
OR
Missed potentially unsafe location in the program
Erroneous conclusion that all locations in the program are safe
OR
OR
Missing VCs
Incomplete program coverage
Missing or incorrect critical annotations
Incomplete or incorrect formalization of safety policy
Incorrect hypotheses at the particular location
Incorrect proof
Incorrect safety obligation
OR
OR
OR
Missing variables
Missing occurrences
Missing paths
Incorrect hypothesis from definition
Hypotheses not maintained correctly along paths
Hypotheses for different paths inconsistent
OR
Definition on incorrect path
Definition has incorrect annotation
Incorrect critical annotation
Incorrect formalization of safety policy
Incorrect safety predicate
VCG constructed incorrect VCs

**Fig. 2.** Fault Tree for Program Verification

Since any location is considered safe if a proof for its corresponding safety obligation can be found, assuming the hypotheses available at that location, the conclusion that the program is safe at all locations can be wrong due to three reasons:

- the proof can be technically wrong (i.e., not conform to the inference rules of the underlying calculus), or
- the safety obligation that is proven can be wrong (i.e., does not imply the safety of the location), or
- the hypotheses used in the proof can be wrong (i.e., do not hold at the location).

Here, we concentrate on the last two reasons and rely on proof checking [29] to mitigate the hazards connected with the first cause. The safety obligation can be wrong if any of the critical annotations are wrong (similar to the case of missing a potentially unsafe location described above), or if the safety policy (including the safety predicate) or its implementation in the VCG are wrong. The hypotheses can be wrong because they have been constructed wrongly at a definition or result from a definition that is on an incorrect path, or because they are not maintained along the paths from the definition to the use, or because the different hypotheses from the different paths are inconsistent to each other.

## 4 Constructing Safety Cases via Annotation Inference

In our work, we consider each violation of the given safety property by the generated code as a hazard. The purpose of the safety case described here is to construct a safety case that argue that the safety property is in fact not violated and thus that the risk associated with this hazard (as identified in section 3) is controlled or mitigated and can not lead to a system failure.

Safety cases [4] are structured arguments, supported by a body of evidence, that provide a convincing and valid case that a system is acceptably safe for a given application in a given operating environment. In our case, the high-level structure of this argument is constructed from information collected by the annotation inference algorithm. However, the evidence still comes from the formal safety proofs. The safety case makes explicit the formal and informal reasoning principles, and reveals the top-level assumptions and external dependencies that must be taken into account. It also provides information about why the generated code can be assumed to be sufficiently safe. It can thus be thought of as "structured reading guide" for the safety proofs and act as a traceable route to the safety requirements, safety claims and evidence that are required to show safety of the generated code.

We use the Goal Structuring Notation [14] as technique to explicitly represent the logical flow of the safety argument. Basically, the safety arguments presented here indicates a linkage between evidence (i.e., formal proofs) and safety claims i.e., that there is no violation to the given safety property that lead to the incorrect formal proofs, and thus the code is indeed safe with respect to the initialization before use safety property. Here, we provide a simplified overview of this safety case. We concentrate on its generic structure and describe its different tiers. We further concentrate on the program itself, leaving the remaining elements (i.e., the formal framework, the certification system and its individual components, and the safety proofs) of the combined safety case for future work.

### 4.1 Tier I: Explaining the Safety Notion

Figure 3 shows the the top tier of the safety case. It starts with the top-level safety goal (i.e., the safety of the generated code with respect to the safety property of interest) and shows how this is achieved by a defensible argument based on the partial correctness of the generated code. The argument stresses the meaning of the Hoare-style framework, specialized to the given safety property. However, the argument structure remains independent of the property. Here, contexts explain the informal interpretation of key notions like "safe" and "safety property". Constraints outline limitations of the approach, in particular, the fact that certification works on an intermediate representation of the source code and only shows a single property, e.g., init-before-use. Hyperlinks refer to additional evidence in the form of documents containing, for example, the model from which the source code has been generated.

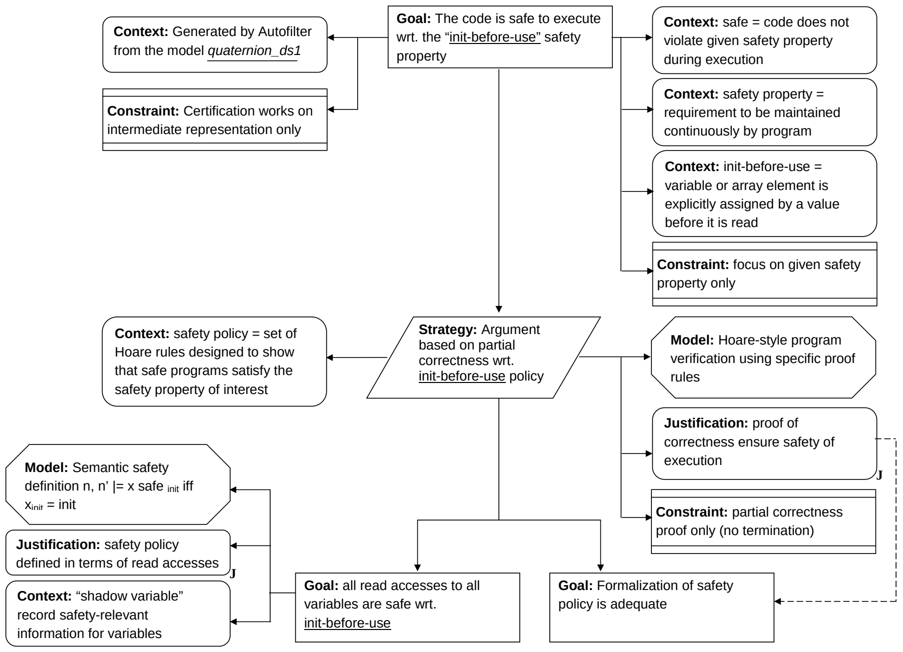
Goal: The code is safe to execute wrt. the "init-before-use" safety property
Context: Generated by Autofilter from the model quaternion_ds1
Constraint: Certification works on intermediate representation only
Context: safe = code does not violate given safety property during execution
Context: safety property = requirement to be maintained continuously by program
Context: init-before-use = variable or array element is explicitly assigned by a value before it is read
Constraint: focus on given safety property only
Context: safety policy = set of Hoare rules designed to show that safe programs satisfy the safety property of interest
Strategy: Argument based on partial correctness wrt. init-before-use policy
Model: Hoare-style program verification using specific proof rules
Justification: proof of correctness ensure safety of execution
J
Constraint: partial correctness proof only (no termination)
Model: Semantic safety definition n, n' |= x safe init iff x init = init
Justification: safety policy defined in terms of read accesses
J
Context: "shadow variable" record safety-relevant information for variables
Goal: all read accesses to all variables are safe wrt. init-before-use
Goal: Formalization of safety policy is adequate

**Fig. 3.** Tier I of Derived Safety Case: Explaining the Safety Notion

The key strategy at this tier and its model (i.e., a Hoare-style partial correctness proof using the dedicated proof rules of the init-before-use safety policy) as well as its limitations (i.e., no termination proof) are made explicit. The strategy reduces showing the safety of the whole program to showing the safety of all read accesses, which emerges as first subgoal. This is justified by the fact that the safety property is defined in terms of variable read accesses. The subgoal is further elaborated by a model of the semantic safety definition, which exactly defines what is meant by "safe", using the notion of shadow variables given as context. The strategy's second subgoal is to show that the safety policy adequately represents the safety property, which is also the foundation of the strategy's original justification (i.e., the claim that the proofs ensure the safe execution of the program). This subgoal is not elaborated further in this safety case but leads to the complementary safety case for the safety logic.

## 4.2 Tier II: Arguing over the Variables

The second tier reduces the safety of all variables in two steps, first to the safety of each individual variable (justified by the fact that the safety property is defined on individual variables) and then to the safety of the individual occurrences. Note that the number of subgoals of both strategies (see Figure 4 for the goal structure) and the safety conditions are program-specific. This information is provided by the annotation inference.

Both strategies are predicated on the assumption that they iterate over the complete list of variables (resp. occurrences). Each individual occurrence then leads to a subgoal to show that the computed safety condition is valid at the location of the variable's occurrence. This reduction to a formal proof obligation is justified by the soundness and completeness of the safety policy; in addition, the specific form of the safety condition is also justified. Note that some of the root cause identified in the fault tree remain as assumptions in the safety case (i.e., the list of variables and their occurrences are assumed to be complete). However, these can be checked easily, since they require no deep analysis of the generated code; in fact, the check could be automated easily.

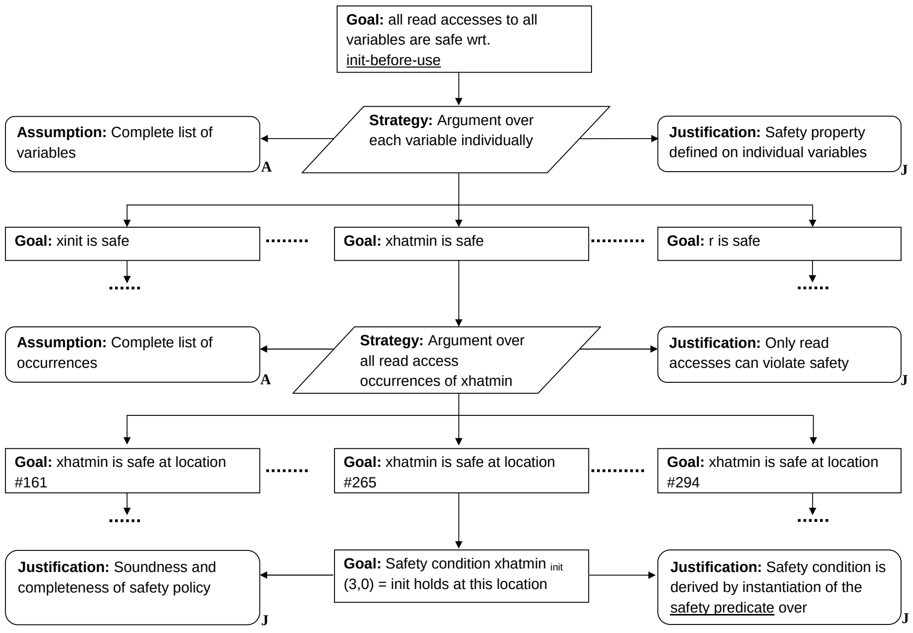
Goal: all read accesses to all variables are safe wrt. init-before-use
Strategy: Argument over each variable individually
Assumption: Complete list of variables
A
Justification: Safety property defined on individual variables
J
Goal: xinit is safe
Goal: xhatmin is safe
Goal: r is safe
Strategy: Argument over all read access occurrences of xhatmin
Assumption: Complete list of occurrences
A
Justification: Only read accesses can violate safety
J
Goal: xhatmin is safe at location #161
Goal: xhatmin is safe at location #265
Goal: xhatmin is safe at location #294
Goal: Safety condition xhatmin init (3,0) = init holds at this location
Justification: Soundness and completeness of safety policy
J
Justification: Safety condition is derived by instantiation of the safety predicate over
J

**Fig. 4.** Tier II of Derived Safety Case: Arguing over the Variables

## 4.3 Tier III: Arguing over the Paths

The final tier (see Figure 5 for the goal structure) argues the safety of each individual variable access, using a strategy based on establishing and maintaining appropriate invariants. This directly reflects the course the annotation inference has taken through the code. The first subgoal is thus to show that the variable safety is established on all paths leading to the current location, using an argument over all definition locations. Here, the model for the subgoal corresponds to the pattern that was applied during annotation inference to identify the definition. Each definition thus leads to a corresponding subgoal and then further to any number of VCs, although here only a single VC emerges in

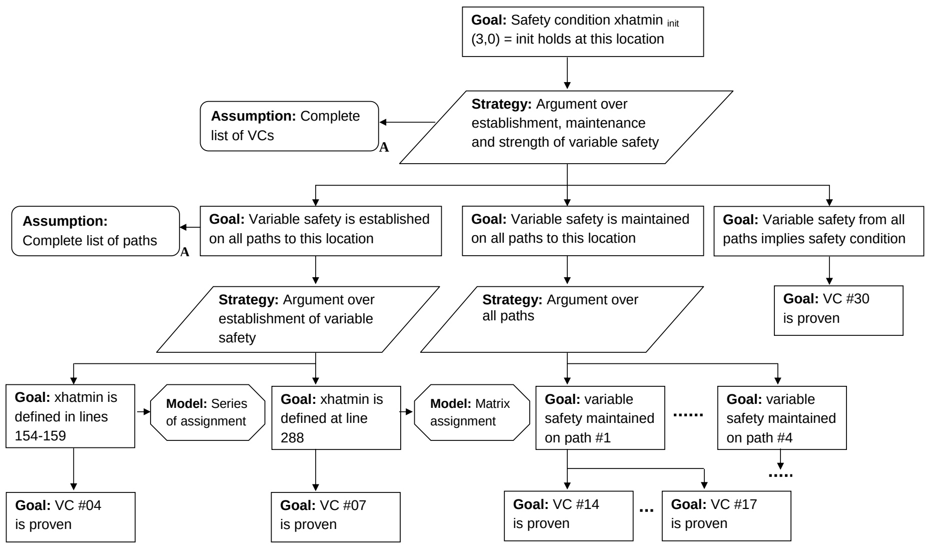
**Goal:** Safety condition xhatmin init (3,0) = init holds at this location
**Strategy:** Argument over establishment, maintenance and strength of variable safety
**Assumption:** Complete list of VCs
A
**Assumption:** Complete list of paths
A
**Goal:** Variable safety is established on all paths to this location
**Goal:** Variable safety is maintained on all paths to this location
**Goal:** Variable safety from all paths implies safety condition
**Strategy:** Argument over establishment of variable safety
**Strategy:** Argument over all paths
**Goal:** VC #30 is proven
**Goal:** xhatmin is defined in lines 154-159
**Model:** Series of assignment
**Goal:** xhatmin is defined at line 288
**Model:** Matrix assignment
**Goal:** variable safety maintained on path #1
**Goal:** variable safety maintained on path #4
**Goal:** VC #04 is proven
**Goal:** VC #07 is proven
**Goal:** VC #14 is proven
**Goal:** VC #17 is proven

**Fig. 5.** Tier III of Derived Safety Case: Arguing over the Paths

both cases. The proof from these VCs demonstrate that the risk identified in the hazard analysis (cf. Figure 2) does not occur for the given program.

Goals that concern properties of the program (e.g., "xhatmin is defined") are decomposed into subgoals that comprise program-independent tasks for the prover, i.e., VCs. The validity of the construction of the VCs depends on the soundness of the rules of the VCG, the simplifier, and the definition of the safety policy, while the correspondence to program locations is based on tracing information added by the VCG and retained during the certification process. We have omitted these details from the safety case.

The second subgoal of the top-level strategy is to show that the established variable safety is maintained along all paths. This proceeds accordingly and the VCs again demonstrate that the identified risk is mitigated. The final subgoal is then to show that the variable safety implies the validity of the safety condition. This can again lead to any number of VCs. If (and only if) all VCs can be shown to hold, then the safety property holds for the entire program. The evidence for the VCs is provided by the formal proofs; we plan to convert these into safety cases as well.

## 5 Related Work

Most previous work on assurance for automated code generation has focused on techniques to ensure the correctness of the code generator. Whalen *et al.* [27] describe a minimum set of requirements for creating code generators that are fit for application in safety-critical systems. However, this set includes a formal correctness proof of the translation implemented by the generator (more precisely, an equivalence proof between model and generated code), which practically feasible only for generators with very similar input and output notations. Our approach, in contrast, is applicable for a much wider range of generators. Stürmer *et al.* [24, 25] present a systematic testing approach and safeguarding techniques for model-based code generation tools. However, the effort easily becomes excessive and testing on its own is insufficient to provide enough assurance for safety-critical systems. Instead, some other basis is required to trust automatically generated code. Both O'Halloran [17] and Denney *et al.* [11] thus suggest that there should be explicit proofs for the correctness of the generated code rather than just trust the correctness of the generator itself.

Only program verification can prove that of program is free of certain defects or does have a certain property of interest. Traditionally, program verification concentrates on showing full functional equivalence between specifications and programs, as for example the KIV system [19]. Necula [16] introduced proof-carrying code (PCC) as new technique to formally verify untrusted code based on specific safety property. PCC and related verification techniques (including our certification approach) generate a large amount of formal mathematical proofs, which cannot be easily understood by users. Consequently, the proofs only tell whether a program is safe or unsafe, but offer no insight into or explanation of the underlying reason. Only few tools combine program verification and documentation, for example the PolySpace static analysis tool [18]. It analyzes programs for compliance with fixed notions of safety, and produces a marked-up browsable program together with a safety report as an Excel spreadsheet. However, unlike our approach, PolySpace does not describe the construction of the underlying safety claims or their relation to the program.

Hughes [13] argues that explanations are appropriate only when we are seeking to understand why something occurred while arguments are appropriate when we want to show that something is true. The argumentation (i.e., safety cases [4, 14]) has been adopted across many industries especially in safety-critical systems. For example, Weaver [26] presents arguments that reflect the contribution of software to critical system safety and Reinhardt [20] presents arguments over the application of the C++ programming language in safety-critical systems. Audsley *et al.* [2] present arguments over the correctness of specification mapping from system model to code and subsequent translation into code. In [12], Galloway *et al.* present a generic argument for technology substitution i.e., argue for the safety of substitution of testing with proof-based verification in the context of certification standards like DO-178B [21]. They present an argument on how can we reasonably conclude, from the evidence available, that the replacement technology is at least as convincing as the evidence produced by testing and there is no impact on system safety. All of this work remains completely generically. While our approach uses a generic argument over safety of the generated code with respect to the safety property of interest, it then shows how this is achieved for the specific code, by constructing a specific defensible argument based on the partial correctness of the generated code. However, our approach remains independent of the given safety property and program, and consequently also independent of the underlying code generator.

## 6 Conclusions

We believe formal methods such as formal software safety certification can provide the highest level of assurance of the code's safety, and have described an approach whereby the inference of annotations drives both formal safety proofs and the construction of a safety case. Here, assurance is not implied by the trust in the generator but follows from an explicitly constructed argument for the generated code.

However, the proofs by themselves are no panacea, and it is important to make explicit which claims are actually proven, and on which assumptions and reasoning principles both the claim and the proof rest. We believe that purely technical solutions such as proof checking [29] fall short of the assurance provide by our safety case, since they do not take into account the reasoning that goes into the construction of the VCs. Here, we use formal proofs only as evidence and base the argumentation structure derived from the course the annotation inference has taken through the code. We consider the safety case as a first step towards a fully-fledged software certificate management system [7].

The work we have described here is still in progress. So far, we have developed the overall structure of the generic program safety case and instantiated it manually. The example shown here uses code generated by our AutoFilter system [28], but the underlying annotation inference algorithm has also been applied to code generated from Matlab models using Real-Time Workshop, and we are confident that the same derivation can be applied there as well. Future work will focus on complementary safety cases that argue the safety of the certification framework itself, in particular the safety of the underlying safety logic (the language semantics and the safety policy) with respect to the safety property (i.e., safety claims) and the safety of other certification components such as the domain theory and the theorem prover.

We believe that the result of our research will be a combined safety case (i.e., for the program being certified, as well as the safety logic and the certification system) that will clearly communicate the safety claims, key safety requirements, and evidence required to trust the generated code.

**Acknowledgements.** This material is based upon work supported by NASA under awards NCC2-1426 and NNA07BB97C. The first author is funded by the Malaysian Government, IPTA Academic Training Scheme.

## References

1. ASCE home page (2007), http://www.adelard.com/web/hnav/ASCE
2. Audsley, N. C., Bate, I. J., Crook-Dawkins, S. K.: Automatic Code Generation for Airborne Systems. In: Proc. of the IEEE Aerospace Conference, pp. 11. IEEE (2003)
3. Basir, N., Denney, E., Fischer, B.: Deriving Safety Cases for the Formal Safety Certification of Automatically Generated Code. In : Huhn, M., Hungar, H., (eds), SafeCert 2008 Intl. Workshop on the Certification of Safety-Critical Software Controlled Systems, ENTCS. Elsevier (2008)
4. Bishop, P., Bloomfield, R.: A methodology for safety case development. In: Redmill, F., Anderson, T. (eds), Industrial Perspectives of Safety-critical Systems: Proc. 6th Safety-critical Systems Symposium, pp. 194-203. Springer (1998)
5. Czarnecki, K., Eisenecker, U. W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
6. Denney, E., Fischer, B.: Correctness of source-level safety policies. In: Araki, K., Gnesi, S., Mandrioli, D., (eds), Proc. FM 2003: Formal Methods, LNCS 2805, pp. 894–913. Springer (2003)

7. Denney, E., Fischer, B.: Software certification and software certificate management systems (Position paper). In: Proc. ASE Workshop on Software Certificate Management Systems, pp. 1-5. ACM (2005)
8. Denney, E., Fischer, B.: A generic annotation inference algorithm for the safety certification of automatically generated code. In: Jarzabek, S., Schmidt, D. C., Veldhuizen, T. L., (eds), Proc. Conf. Generative Programming and Component Engineering, pp. 121-130. ACM (2006)
9. Denney, E., Fischer, B.: Annotation inference for safety certification of automatically generated code (extended abstract). In: Uchitel, S., Easterbrook, S., (eds), Proc. 21st ASE, pp. 265-268. IEEE (2006)
10. Denney, E., Trac, S.: A Software Safety Certification Tool for Automatically Generated Guidance, Navigation and Control Code. In: Electronic Proc. IEEE Aerospace Conference. IEEE (2008)
11. Denney, E., Fischer, B.: Certifiable program generation. In: Proc.Conf. Generative Programming and Component Engineering, LNCS 3676, pp. 17-28. Springer (2005)
12. Galloway, A., Paige, R.F., Tudor, N. J., Weaver, R. A, Toyn, I., McDermid, J.: Proof vs testing in the context of safety standards. In:The 24th Digital Avionics Systems Conference, vol. 2., pp. 14. IEEE Press (2005)
13. Hughes, W.: Critical Thinking. Broadview Press (1992)
14. Kelly, T. P.: Arguing safety a systematic approach to managing safety cases. PhD Thesis, University of York (1998)
15. Leveson, N. G.: Safeware: System Safety and Computers. Addison-Wesley (1995)
16. Necula, G. C.: Proof-carrying code. In: Proc. 24th Conf. Principles of Programming Languages, pp. 106-119. ACM (1997)
17. O'Halloran, C.: Issues for the automatic generation of safety critical software. In: Proc.15th Conf. Automated Software Engineering, pp. 277-280. IEEE (2000)
18. PolySpace Technologies, http://www.polyspace.com
19. Reif, W.: The KIV Approach to Software Verification. In: KORSO: Methods, Languages and Tools for the Construction of Correct Software, LNCS 1009, pp. 339-370. Springer (1995)
20. Reinhardt, D. W.: Use of the C++ Programming Language in Safety Critical Systems. Master Thesis, University of York (2004)
21. RTCA, "Software Considerations in Airborne Systems and Equipment Certification". RTCA (1992)
22. Smith, D. R.: KIDS: A semi-automatic program development system. IEEE Trans. on Software Engineering, vol. 16(9), pp. 286-290. IEEE (1990)
23. Stickel, M., Waldinger, R., Lowry, M., Pressburger, T., Underwood, I.: Deductive composition of astronomical software from subroutine libraries. In: Proc. 12th Conf. Automated Deduction, LNAI 814, pp. 341-355. Springer (1994)
24. Stürmer, I., Conrad, M.: Test suite design for code generation tools. In: Proc. 18th Conf. Automated Software Engineering, pp. 286-290. IEEE (2003)
25. Stürmer, I., Weinberg, D., Conrad, M.: Overview of Existing Safeguarding Techniques for Automatically Generated Code. In: Proc. of 2nd Intl. ICSE Workshop on Software Engineering for Automotive Systems, pp. 1-6. ACM (2006)
26. Weaver, R. A.: The Safety of Software–Constructing and Assuring Arguments. PhD Thesis, University of York (2003)
27. Whalen, M. W., Heimdahl, M. P.E.: On the requirements of High-Integrity Code Generation. In: Proc. 4th High Assurance in Systems Engineering Workshop, pp. 217-224. IEEE (1999).
28. Whittle, J., Schumann, J.: Automating the implementation of Kalman filter algorithms. ACM Transactions on Mathematical Software, vol. 30(4), pp. 434-453. ACM (2004)
29. Wong, W.: Validation of HOL proofs by proof checking. Formal Methods in System Design: An International Journal, vol. 14, pp. 193-212. Kluwer Academic Publishers (1999)