Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 238 (2009) 19–26
www.elsevier.com/locate/entcs

Deriving Safety Cases for the Formal Safety Certification of Automatically Generated Code

Nurlida Basir [1]

ECS, Southampton University
Southampton, SO17 1BJ, UK

Ewen Denney [2]

USRA/RIACS, NASA Ames Research Center
Mountain View, CA 94035, USA

Bernd Fischer [3]

ECS, Southampton University
Southampton, SO17 1BJ, UK

Abstract

We present an approach to systematically derive safety cases for automatically generated code from information collected during a formal, Hoare-style safety certification of the code. This 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; however, the evidence still comes from the formal safety proofs. It uses a generic goal-based argument that is instantiated with respect to the certified safety property (i.e., safety claims) and the program. This will be combined with a complementary safety case that argues the safety of the framework itself, in particular the correctness of the Hoare rules with respect to the safety property and the trustworthiness of the certification system and its individual components.

Keywords: Automated code generation, Hoare logic, formal code certification, safety case, Goal Structuring Notation.

[1] Email: nb206r@ecs.soton.ac.uk. Supported by the IPTA Academic Training Scheme of the Ministry of Higher Education of the Malaysian Government.

[2] Email: Ewen.W.Denney@nasa.gov. Supported by NASA under awards NCC2-1426 and NNA07BB97C.

[3] Email: b.fischer@ecs.soton.ac.uk.

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2009.09.003

1 Introduction

Model-based design and automated code generation have become popular, but substantial obstacles remain to their more 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 [3] can be used to demonstrate the required safety and integrity of the generated code, 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. The complexity of the tools used can lead to unforeseen interactions and thus causes additional concerns about the trustworthiness of the assurance claims. 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.

Here, we address this problem and present an approach currently under development to systematically (and ultimately automatically) derive safety cases from information collected during the formal 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 generic, multi-tiered safety case that is instantiated with respect to a given safety property and program. Its upper tier simply instantiates the notion of safety and the formal definitions for the given safety property while its 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 argument structure directly follows the course the annotation construction takes through the program. Our approach is thus independent of the given safety property and program, and consequently also independent of the underlying code generator. These three tiers together constitute a single safety case that justifies the safety of the program. This paper discusses the structure of this safety case. It will eventually be complemented by an additional safety case that justifies the trustworthiness of the certification tool and framework itself. This 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.

We believe that the combined safety case (i.e., for the program being certified, as well as the safety logic and the certification system) will clearly communicate the safety claims, key safety requirements, and evidence required to trust the generated code. We expect that this will alleviate distrust in code generators, which remains a problem for their use in safety-critical applications.

This paper describes work still in progress. So far we have developed the overall

structure of the generic program 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].

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 software 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) [3]. 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 a theorem prover) the VCs to the complementary 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 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 safety property.

Our example below uses initialization safety but our framework can handle a variety other safety properties including absence of out-of-bounds array accesses [3]; we expect that other properties handled by proof-carrying code such as null pointer dereferences [7] 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.

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. The purpose of annotation inference [5,6] 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 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 the initialization property, an invariant on a for-loop might express that an array has been initialized up to the loop index ($\forall j \leq i \cdot A_{\text{init}}[j]$). 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 (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.

4 Deriving Safety Cases via Annotation Inference

In our work, we consider each violation of the given safety property as a hazard. To demonstrate that this hazard can not lead to a system failure, we construct a safety case that argues that the safety property is in fact not violated and thus that the risk associated with this hazard is controlled or mitigated. Safety cases 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 [2]. 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 only makes explicit the formal and informal reasoning principles, and reveals the top-level assumptions and external dependencies that must be taken into account. It can thus be thought of as "structured reading guide" for the safety proofs.

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

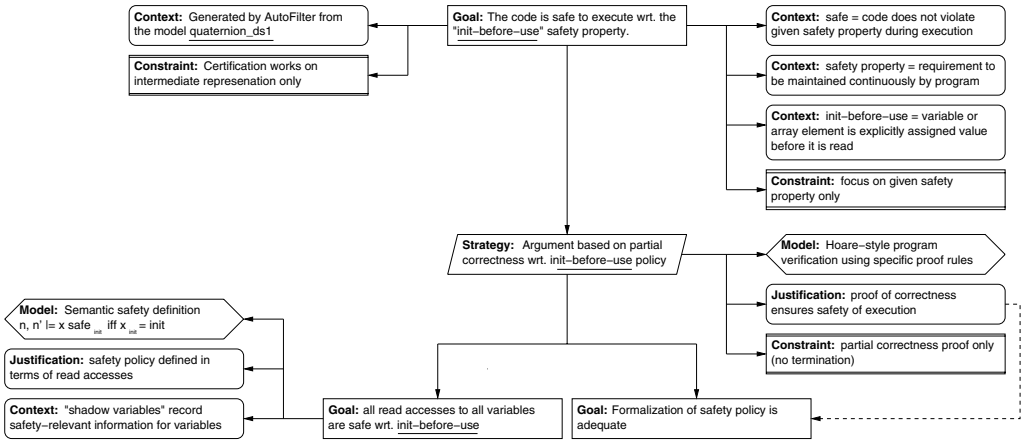

Fig. 1. Tier I of Derived Safety Case: Explaining the Safety Notion

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 1 shows the goal structure for 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 formal 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. It uses contexts to explain the informal interpretation of key notions like "safe" and "safety property" and constraints to outline limitations of the approach, in particular the fact the certification works on an intermediate representation of the source code and only shows a single property. Hyperlinks refer to additional evidence in the form of documents containing, for example, the model from which the source code has been generated.

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.

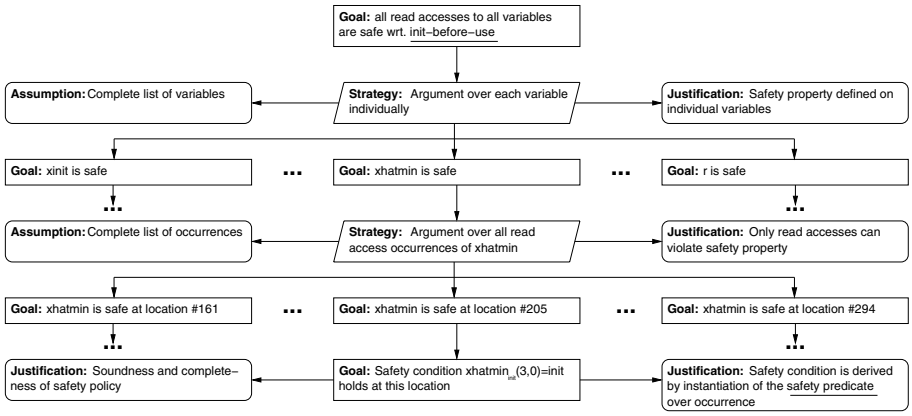

Fig. 2. Tier II of Derived Safety Case: Arguing over the Variables

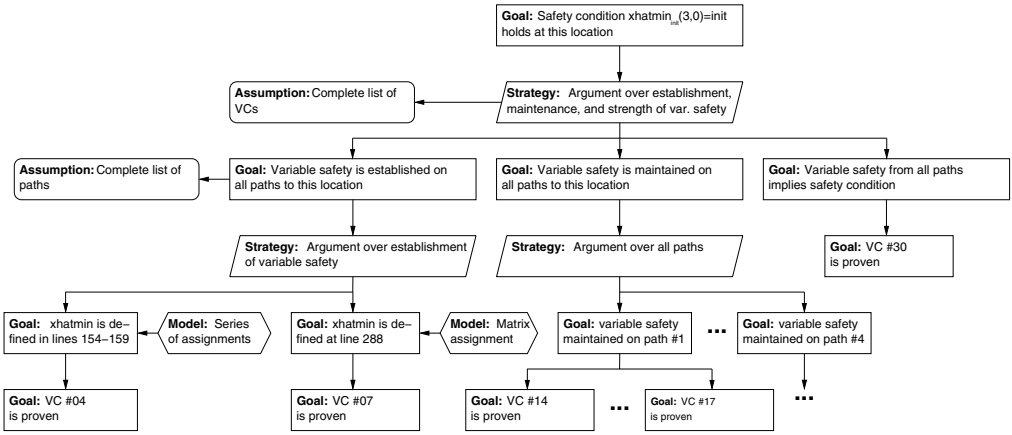

Fig. 3. Tier III of Derived Safety Case: Arguing over the Paths

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 2 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.

4.3 Tier III: Arguing over the Paths

The final tier (see Figure 3 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 a formal 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 both cases.

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 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. 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 Conclusions

Software development standards for safety-critical domains such as DO-178B [8] are typically process-oriented and require that code generators are qualified for application, often using an elaborate testing regime [9]. This time-consuming and expensive process slows down generator development and application. We believe that product-oriented assurance approaches are a viable alternative. Here, assurance is not implied by the trust in the generator but follows from an explicitly constructed argument for the generated code. We further believe that formal methods such as formal software safety certification can provide the highest level of assurance of the code's safety and integrity, and have described an approach whereby the inference of annotations drives both formal safety proofs and the construction of a safety case.

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 [11] 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. In fact, we consider the safety case only as a first step towards a fully-fledged software certificate management system [4].

We have described work 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 [10], but the underlying annotation inference algorithm has also been applied to code generated from Matlab models using Real-Time Workshop, and we expect that the same derivation can be applied there as well. Current work involves constructing a more comprehensive, combined safety case that covers the components of the certification system itself (i.e., the formal framework, the inference system and its individual components, and the safety proofs). There we rely on the fact that trust in the complex components of the system can be reduced to trust in simpler components. For example, the use of proof checking mitigates the risk of the automated theorem prover.

References

[1] *ASCE home page* (2007), `www.adelard.com/web/hnav/ASCE`.

[2] Bishop, P. and R. Bloomfield, *A methodology for safety case development*, in: F. Redmill and T. Anderson, editors, *Industrial Perspectives of Safety-critical Systems: Proceedings of the Sixth Safety-critical Systems Symposium* (1998), pp. 194–203.

[3] Denney, E. and B. Fischer, *Correctness of source-level safety policies*, in: K. Araki, S. Gnesi and D. Mandrioli, editors, *Proc. FM 2003: Formal Methods*, LNCS **2805** (2003), pp. 894–913.

[4] Denney, E. and B. Fischer, *Software certification and software certificate management systems (position paper)*, in: *Proceedings of the ASE Workshop on Software Certificate Management Systems (SoftCeMent '05)*, 2005, pp. 1–5.

[5] Denney, E. and B. Fischer, *Annotation inference for safety certification of automatically generated code (extended abstract)*, in: S. Uchitel and S. Easterbrook, editors, *Proc. 21st ASE* (2006), pp. 265–268.

[6] Denney, E. and B. Fischer, *A generic annotation inference algorithm for the safety certification of automatically generated code*, in: S. Jarzabek, D. C. Schmidt and T. L. Veldhuizen, editors, *Proc. Conf. Generative Programming and Component Engineering* (2006), pp. 121–130.

[7] Necula, G. C., *Proof-carrying code*, in: *Proc. 24th POPL* (1997), pp. 106–19.

[8] RTCA, "Software Considerations in Airborne Systems and Equipment Certification," RTCA, 1992.

[9] Stürmer, I. and M. Conrad, *Test suite design for code generation tools*, in: *Proceedings of 18th IEEE International Conference on Automated Software Engineering* (2003), pp. 286–290.

[10] Whittle, J. and J. Schumann, *Automating the implementation of Kalman filter algorithms*, ACM Transactions on Mathematical Software **30** (2004), pp. 434–453.

[11] Wong, W., *Validation of HOL proofs by proof checking*, Formal Methods in System Design: An International Journal **14** (1999), pp. 193–212.