Deriving Safety Cases from Machine-Generated Proofs

Nurlida Basir and Bernd Fischer
ECS, University of Southampton
SO17 1BJ, U.K
(nb206r,b.fischer)@ecs.soton.ac.uk

Ewen Denney
SGT, NASA Ames Research Center
Mountain View, CA 94035, U.S.A
Ewen.W.Denney@nasa.gov

Abstract

Proofs provide detailed justification for the validity of claims and are widely used in formal software development methods. However, they are often complex and difficult to understand, because they use machine-oriented formalisms; they may also be based on assumptions that are not justified. This causes concerns about the trustworthiness of using formal proofs as arguments in safety-critical applications. Here, we present an approach to develop safety cases that correspond to formal proofs found by automated theorem provers and reveal the underlying argumentation structure and top-level assumptions. We concentrate on natural deduction proofs and show how to construct the safety cases by covering the proof tree with corresponding safety case fragments.

1 Introduction

Demonstrating the safety of large and complex software-intensive systems requires marshalling large amounts of diverse information, e.g., models, code or mathematical equations and formulas. Obviously tools supported by automated analyses are needed to tackle this problem. For the highest assurance levels, these tools need to produce a *traceable safety argument* that shows in particular where the code as well as the argument itself depend on any external assumptions but many techniques commonly applied to ensure software safety do not produce enough usable evidence (i.e., justification for the validity of their claims) and can thus not provide any further insights or arguments. In contrast, in formal software safety certification [3], formal proofs are available as evidence. However, these proofs are typically constructed by automated theorem provers (ATPs) based on machine-oriented calculi such as resolution [10]. They are thus often too complex and too difficult to understand, because they spell out too many low-level details. Moreover, the proofs may be based on assumptions that are not valid, or may contain steps that are not justified. Consequently, concerns remain about using these proofs as *arguments* rather than just *evidence* in safety-critical applications. In this paper we address these concerns by systematically constructing safety cases that correspond to formal proofs found by ATPs and explicitly highlight the use of assumptions.

The approach presented here reveals and presents the proof's underlying argumentation structure and top-level assumptions. We work with natural deduction (ND) proofs, which are closer to human reasoning than resolution proofs. We explain how to construct the safety cases by covering the ND proof tree with corresponding safety case fragments. The argument is built in the same top-down way as the proof: it starts with the original theorem to be proved as the top goal and follows the deductive reasoning into subgoals, using the applied inference rules as strategies to derive the goals. However, we abstract away the obvious steps to reduce the size of the constructed safety cases. The safety cases thus provide a "structured reading guide" for the proofs that allows users to understand the claims without having to understand all the technical details of the formal proof machinery. This paper is a continuation of our previous work to construct safety cases from information collected during the formal verification of the code [2], but here we concentrate on the proofs rather than the verification process.

2 Formal Software Safety Certification

Formal software safety certification uses formal techniques based on program logics to show that the program does not violate certain conditions during its execution [3]. A *safety property* is an exact char-

E. Denney, T. Jensen (eds.); The 3rd International Workshop on Proof Carrying Code and Software Certification, pp. 13-17

acterization of these conditions, based on the operational semantics of the programming language. Each safety property thus describes a class of hazards. The safety property is enforced by a *safety policy*, i.e., a set of verification rules that take initial set of safety requirements that formally represent the specific hazards identified by a safety engineer [8], and derive a number of proof obligations. Showing the safety of a program is thus reduced to formally showing the validity of these proof obligations: a program is considered safe wrt. a given safety property if proofs for the corresponding safety proof obligations can be found. Formally, this amounts to showing $D \cup A \models P \Rightarrow C$ for each obligation i.e., the formalization of the underlying *domain theory* D and a set of *formal certification assumptions* A entail a conjecture, which consists of a set of premises P that have to imply the *safety condition* C.

The different parts of these proof obligations have different levels of trustworthiness, and a safety case should reflect this. The hypotheses and the safety condition are inferred from the program by a trusted software component implementing the safety policy, and their construction can already be explained in a safety case [2]. In contrast, both the domain theory and the assumptions are manually constructed artifacts that require particular care. In particular, the safety case needs to highlight the use of assumptions. These have been formulated in isolation by the safety engineer and may not necessarily be justified, and are possibly inconsistent with the domain theory. Moreover, fragments of the domain theory and the assumptions may be used in different contexts, so the safety case must reflect which of them are available at each context. By elucidating the reasoning behind the certification process and drawing attention to potential certification problems, there is less of a need to trust the certification tools, and in particular, the manually constructed artifacts.

3 Converting Natural Deduction Proofs into Safety Cases

Natural deduction [6] systems consist of a collection of proof rules that manipulate logical formulas and transform premises into conclusions. A conjecture is proven from a set of assumptions if a repeated application of the rules can establish it as conclusion. Here, we focus on some of the basic rules; a full exposition of the ND calculus can be found in the literature [6].

Conversion Process. ND proofs are simply trees that start with the conjecture to be proven as root, and have given axioms or assumed hypotheses at each leaf. Each non-leaf node is recursively justified by the proofs that start with its children as new conjectures. The edges between a node and all of its children correspond to the inference rule applied in this proof step. The proof tree structure is thus a representation of the underlying argumentation structure. We can use this interpretation to present the proofs as *safety cases* [7], which are structured arguments as well and represent the linkage between evidence (i.e., the deductive reasoning of the proofs from the assumptions to the derived conclusions) and claims (i.e., the original theorem to be proved). The general idea of the conversion from ND proofs to safety cases is thus fairly straightforward. We consider the conclusion as a goal to be met; the premise(s) become(s) the new subgoal(s). For each inference rule, we define a safety case template that represents the same argumentation. The underlying similarity of proofs and safety cases has already been indicated in [7] but as far as we know, this idea has never been fully explored or even been applied to machine-generated proofs (see Figure 1 for some example rules and templates). Here, we use the Goal Structuring Notation [7] as technique to explicitly represent the logical flow of the proof's argumentation structure.

Implications. The implication elimination follows the general pattern sketched above but in the introduction rule we again temporarily assume A as hypothesis together with the list of other available hypotheses, rather than deriving a proof for it. We then proceed to derive B, and *discharge* the hypothesis by the introduction of the implication. The hypothesis A can be used at given in the prove of B, but the conclusion $A \Rightarrow B$ no longer depends on the hypothesis A after B has been proved. In the safety case fragment, we use a justification to record the use of the hypothesis A, and thus to make sure that the

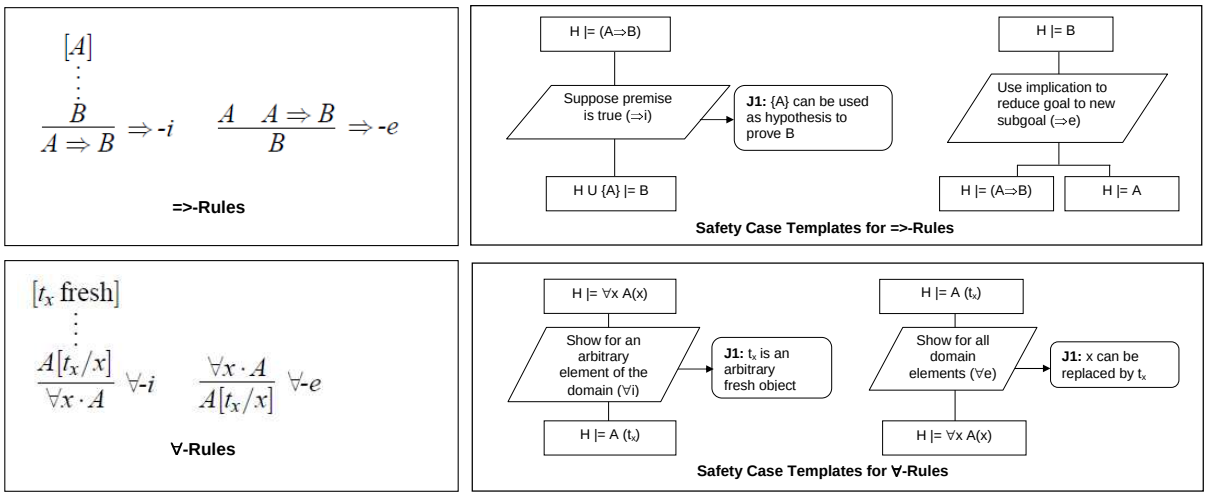

Figure 1: Safety Case Templates for Natural Deduction Rules

introduced hypotheses are tracked properly.

Universal quantifiers. The ND proof rules for quantifiers focus on the replacement of the bound variables with objects and vice versa. For example, in the elimination rule for universal quantifiers, we can conclude the validity of the formula for any chosen domain element t_x. In the introduction rule, however, we need to show it for an arbitary but fresh object t_x (that is, a domain element which does not appear elsewhere in H, A, or the domain theory and assumptions). If we can derive a proof of A, where x is replaced by the object t_x, we can then discharge this assumption by introduction of the quantifier. The safety case fragments record this replacement as justification. The hypotheses available for the subgoals in the $\forall$-rules are the same as those in the original goals.

4 Hypothesis Handling

An automated prover typically treats the domain theory D and the certification assumptions A as premises and tries to derive $\bigwedge(D \cup A) \wedge P \Rightarrow C$ from an empty set of hypotheses. As the proof tree grows, these premises will be turned into hypotheses, using the $\Rightarrow$- introduction rule (see Figure 1). In all other rules, the hypotheses are simply inherited from the goal to the subgoals. However, not all hypotheses will actually be used in the proof, and the safety case should highlight those that are actually used. This is particularly important for the certification assumptions. We can achieve this by modifying the template for the $\Rightarrow$- introduction (see Figure 2a). We can distinguish between the hypotheses that are actually used in the proof of the conclusion (denoted by $A_1, ..., A_k$) and those that are vacuously discharged by the $\Rightarrow$- introduction (denoted by $A_k + 1, .., A_n$). We can thus use two different justifications to mark this distinction. Note that this is only a simplification of the presentation and does not change the structure of the underlying proof, nor the validity of the original goal. It is thus different from using a *relevant implication* [1] under which $A \Rightarrow B$ is only valid if the hypothesis A is actually used.

In order to minimize the number of hypotheses tracked by the safety case, we need to analyze the proof tree from the leaves up, and propagate the hypotheses towards the root. By revealing only these used hypotheses as assumptions, the validity of their use in deriving the proof can be checked more easily. In our work, we also highlight the use of the external certification assumptions that have been formulated in isolation by the safety engineer. For example, in Figure 2b, the hypothesis has_unit(float_7_0e_minus_1, ang_vel), meaning that a particular floating point variable represents an angular velocity, has been speci-

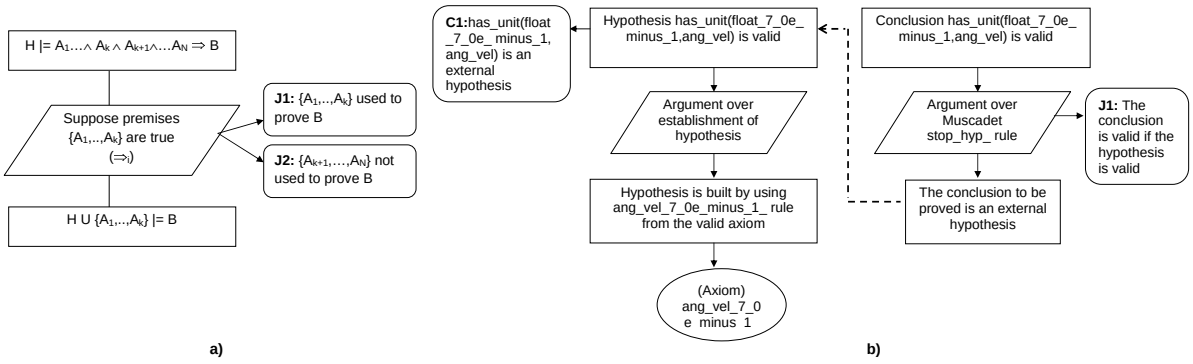

Figure 2: Hypothesis Handling

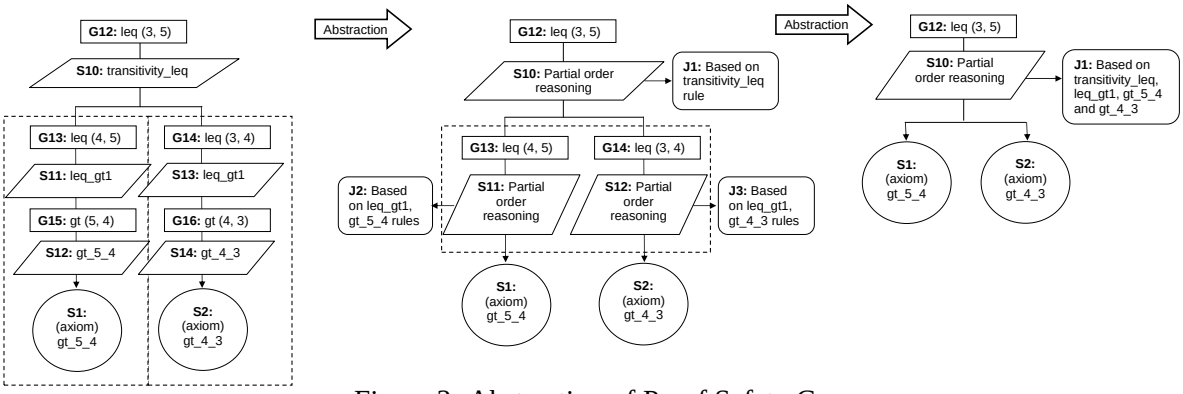

Figure 3: Abstraction of Proof Safety Case

fied as external assumption. This is tracked properly in the safety case, and its role in deriving the proofs can be checked easily.

5 Proof Abstraction

We have applied our approach to proofs found by the Muscadet [9] theorem prover during the formal certification of the frame safety of a component of an attitude control system as an example. Muscadet is based on ND, but to improve performance, it implements a variety of derived rules in addition to the basic rules of the calculus. This includes rules for dedicated equality handling, as well as rules that the system builds from the definitions and lemmas, and that correspond the application of the given definitions and lemmas. While these rules make the proofs shorter, their large number makes the proofs also in turn more difficult to understand. This partially negates the original goal of using a ND prover. We thus plan to optimize the resulting proofs by removing some of the book-keeping rules (e.g., return_proof) that are not central to the overall argumentation structure. Similarly, we plan to collapse sequences of identical book-keeping rules into a single node. In general, however, we try to restructure the resulting proof presentation to help in emphasizing the essential proof steps. In particular, we plan to group sub-proofs that apply only axioms and lemmas from certain obvious parts of the domain theory (e.g., ground arithmetic or partial order reasoning) and represent them as a single strategy application. Figure 3 shows an example of this. Here, the first abstraction step collapses the sequences rooted in G13 and G14, noting the lemmas which had been used as strategies as justifications, but keeping the branching that is typical for the transitivity. A second step then abstracts this away as well.

6 Conclusions

We have described an approach whereby a safety case is used as a "structured reading" guide for the safety proofs. Here, assurance is not implied by the trust in the ATPs but follows from the constructed argument of the underlying proofs. However, the straightforward conversion of ND proofs into safety cases turn out to be far from satisfactory as the proofs typically contain too many details. In practice, a superabundance of such details is overwhelming and unlikely to be of interest anyway so careful use of abstraction is needed [5].

The work we have described here is still in progress. So far, we have automatically derived safety cases for the proofs found by Muscadet prover [9]. This work complements our previous work [2] where we used the high-level structure of annotation inference to explicate the top-level structure of such software safety cases. We consider the safety case as a first step towards a fully-fledged software certificate management system [4]. We also believe that our research will result in a comprehensive safety case (i.e., for the program being certified the safety logic, and the certification system) that will clearly communicate the safety claims, key safety requirements, and evidence required to trust the software safety.

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] A.R. Anderson and N. Belnap. *Entailment: the logic of relevance and necessity*. Princeton University Press, 1975.

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

[3] E. Denney and B. Fischer. Correctness of Source-Level Safety Policies . In *Proc. FM 2003: Formal Methods*, 2003.

[4] E. Denney and B. Fischer. Software Certification and Software Certificate Management Systems (position paper). *Proceedings of the ASE Workshop on Software Certificate Management Systems (SoftCeMent '05)*, pages 1–5, 2005.

[5] E. Denney, J. Power, and K. Tourlas. Hiproofs: A Hierarchical Notion of Proof Tree. In *Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI)*, volume 155, pages 341 – 359, 2006.

[6] M. Huth and M. Ryan. *Logic in Computer Science Modelling and Reasoning about Systems*, volume 2nd Edition. Cambridge University Press, 2004.

[7] T. P. Kelly. *Arguing Safety - A Systematic Approach to Managing Safety Cases*. PhD thesis, University of York, 1998.

[8] N. G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[9] D. Pastre. MUSCADET 2.3: A Knowledge-Based Theorem Prover Based on Natural Deduction. In *IJCAR*, pages 685–689, 2001.

[10] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *ACM*, 1965.