DERIVING SAFETY CASES FROM AUTOMATICALLY CONSTRUCTED PROOFS

Nurlida Basir*, Ewen Denney† and Bernd Fischer*

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

Keywords: formal proofs, safety case, natural deduction, automated theorem provers.

Abstract

Formal 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 the formalism in which they are constructed and encoded is usually machine-oriented, and 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 style proofs, which are closer to human reasoning than resolution proofs, and show how to construct the safety cases by covering the natural deduction proof tree with corresponding safety case fragments. We also abstract away logical book-keeping steps, which reduces the size of the constructed safety cases. We show how the approach can be applied to the proofs found by the Muscadet prover.

1 Introduction

Demonstrating the correctness of large and complex software-intensive systems requires marshalling large amounts of diverse information, including models, code, specifications, mathematical equations and formulas, and tables of engineering constants. Tools supported by automated analyses can be used to produce a *traceable safety argument* [10] that shows in particular where the code, verification artifacts and the argument itself depends on any external assumptions.

However, many tools commonly applied to ensure software safety rely on black-box techniques such as static analysis [22] or model checking [7] that produce only opaque claims about the safety of the software but not enough evidence to justify their claim. They can thus not provide any further insights or arguments. In contrast, in formal software safety certification [8], as in other formal software development methods [3, 5], formal proofs can in principle be used as evidence. Such proofs use mathematical and logical reasoning to show that the software satisfies certain requirements, which typically include program execution safety properties such as absence of array-out-of-bounds violations, as well as functional correctness properties. However, in practice there are reservations about the use of formal proofs as evidence (or even arguments) in safety-critical applications. Concerns that the proofs may be based on assumptions that are not valid, or may contain steps that are not justified, can undermine the reasoning used to support the assurance claim. Moreover, these proofs are typically constructed automatically by automated theorem provers (ATPs) based on machine-oriented calculi such as resolution which are often too complex and too difficult to understand by engineers, because the formalisms spell out too many low-level details. In this paper we address these issues by systematically constructing safety cases that correspond to formal proofs found by ATPs and explicitly call out the use of external assumptions. The safety cases highlight the argument structure underlying the proof construction, and help showing how the truth of the theorem follows from the different assertions and subgoals.

The approach presented here combines abstraction and visualization to reveal and present the proof's underlying argumentation structure and top-level assumptions. We work with natural deduction (ND) style proofs, which are goal-directed and thus closer to human reasoning than resolution proofs, and we show how the approach can be applied to the proofs found by the Muscadet ATP [21]. 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 ATP's book-keeping steps, which 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 [4], but here we concentrate on the certification components, i.e., the domain theory and the ATP used to support the software safety assurance process.

2 Formal Software Safety Certification

Our work is set in the context of formal software safety certification [8], where we use formal source code analysis 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. In our work we consider each violation of the safety property a potential of hazard. Each safety property thus describes a class of hazards. In our framework, the safety property is enforced by a *safety policy*, i.e., a set of verification rules that derived from initial set of safety requirements that formally represent the specific property identified by a safety engineer, and derive a number of logical 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

$$D \cup A \models P \Rightarrow C \qquad (1)$$

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 *preconditions* P that have to imply the *safety condition* C. The domain theory formalizes the extra-logical operations that occur in the obligations; it includes arithmetic functions and relations, programming language operations such as array indexing as well as application-specific operations such as matrix inversion. Assumptions typically specify global properties required by the component (e.g., the physical units of the input signals), while preconditions and safety conditions refer to properties at intermediate locations in the code.

The different elements of these proof obligations have different origins, and thus different levels of trustworthiness, and a safety case should reflect this. The premises 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 [4]. In contrast, both the domain theory and the assumptions are manually constructed artifacts that require particular care. The main hazard that we address in the safety cases here, by making explicit the use of hypotheses, is the unintended introduction of logical inconsistencies that can be exploited by the ATPs to construct logically correct but vacuous proofs.

The necessary analysis is hampered by the fact that the prover does not work on the obligations in the form as given in (1) but uses the form

$$\models \bigwedge(D \cup A) \wedge P \Rightarrow C \qquad (2)$$

which is logically equivalent but blurs the distinction between domain theory, assumptions, and preconditions, and lumps them all together as logical premises. However, proofs typically use only a subset of these premises as hypotheses, and the safety case should make explicit those that are actually used. In particular, it needs to highlight the use of assumptions. These have been formulated in isolation by the safety engineer and may not necessarily be justified, and the possibility of a logical inconsistency with the domain theory is substantially higher. 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 Arguments

3.1 Natural Deduction

Natural deduction [14, 15] is a form of proof that attempts to provide a foundational yet intuitive system to construct formal proofs. It consists 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. The proof rules can be divided into basic rules, derived rules (which can be seen as proof “macros” that group together multiple inference steps) and replacement rules (which are derived rules for equivalence and equality handling). Here, we focus on some of the basic rules; a full exposition of natural deduction can be found in the literature [17].

Natural deduction uses two sets of rules for each logical connective or quantifier ($\wedge, \vee, \ldots, \forall, \ldots$), where one *introduces* the symbol, while the other *eliminates* it. In the introduction rules, the connective or quantifier is used as the top-level operator symbol of the unique conclusion, while it occurs in the elimination rules in the same role in one of the premises.

3.2 Conversion Process

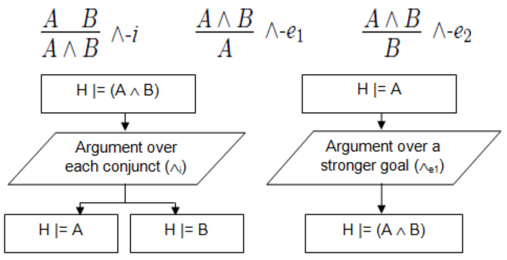
Natural deduction proofs are simply trees that start with the conjecture to be proven as root, and have given axioms or assumed hypotheses at each leaf. Each non-leaf node is recursively justified by the proofs that start with its children as new conjectures. The edges between a node and all of its children 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* [18], 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 and the premise(s) as a subgoal(s); we further consider the applied inference rule as the strategy that shows *how* the conclusion is met. 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

[18] but as far as we know, this idea has never been fully explored or even been applied to machine-generated proofs.

The conversion we present here preserves the inferences and formulas of the original proof, but avoids overloading the constructed arguments with trivial proof steps. We identify semantically related or repeated identical inferences that can be abstracted away in order to construct a more concise argument. In the following we describe the safety case templates for the base rules of the calculus. We use the Goal Structuring Notation [18] to explicitly represent the logical flow of the proof's argumentation structure.

3.3 Conjunctions

The rules for conjunction introduction and elimination shown in Figure 1 directly represent the intuitive interpretation of conjunctions: if A is true and B is true, then evidently A∧B is true as well (∧-i), and if A∧B is true, then both A and B must be as well (∧-e1 resp. ∧-e2). The hypotheses that are available to show A∧B true are also available to show A (resp. B) true as well. Similarly in the case of ∧-e1 (resp. ∧-e2) where the available hypotheses to show each conjunct true are also available to show A∧B true.



$\frac{A \quad B}{A \wedge B}$ ∧-i
$\frac{A \wedge B}{A}$ ∧-e_1
$\frac{A \wedge B}{B}$ ∧-e_2
H |= (A ∧ B)
Argument over each conjunct (∧i)
H |= A
H |= B
H |= A
Argument over a stronger goal (∧e1)
H |= (A ∧ B)

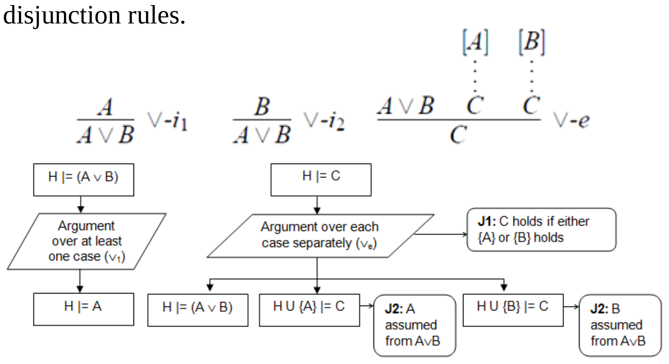
Figure 1: Safety Case Templates for ∧-Rules.

The ∧-rules can be directly converted into safety cases. In the case of ∧-introduction, the satisfaction of the conclusion (i.e., goal of the safety case) is implied by the satisfaction of the two premises (i.e., subgoals of the safety case) based on the strategy of ∧-introduction rule. For the ∧-elimination rule, the strategy shows a logically stronger goal we can conclude A (resp.B) if we have a proof of A∧B. Figure 1 shows the safety case fragments for the conjunction rules.

3.4 Disjunctions

A disjunction can be introduced as long as one of the disjuncts is already known i.e., if A (resp. B) is true, then evidently A∨B is true as well. In the safety case, a goal A∨B is constructed, which is justified by the subgoal A (resp. B) via the strategy (∨-i_1) (resp. ∨-i_2). The hypotheses that are available to show A∨B true are also available to show subgoal A (resp. B) true.

In contrast, in disjunction elimination, we only know that A∨B holds, but not which of A or B is true, so that we need to reason by cases to conclude C from A∨B, i.e., separately consider each of the two cases for the disjunction to be true. In the first case we thus assume A together with the available hypotheses and try to derive C, in the second case we assume B together with the available hypotheses and try to derive C. If both cases succeed, we can conclude C. The safety case fragment makes this argument explicit, and, in particular, explicitly justifies the use of the respective assumptions in the two cases. Figure 2 shows the safety case fragments for the disjunction rules.

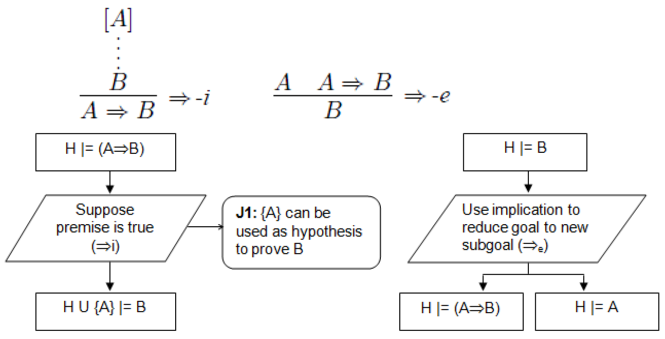


$\frac{A}{A \vee B}$ ∨-i_1
$\frac{B}{A \vee B}$ ∨-i_2
$\frac{A \vee B \quad \begin{matrix}[A]\\ \vdots \\ C\end{matrix} \quad \begin{matrix}[B]\\ \vdots \\ C\end{matrix}}{C}$ ∨-e
H |= (A ∨ B)
Argument over at least one case (∨i)
H |= A
H |= C
Argument over each case separately (∨e)
J1: C holds if either {A} or {B} holds
H |= (A ∨ B)
H U {A} |= C
J2: A assumed from A∨B
H U {B} |= C
J2: B assumed from A∨B

Figure 2: Safety Case Templates for ∨-Rules.

3.5 Implications

The implication elimination follows the standard pattern 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 in the proof of B, but the conclusion A=>B no longer depends on the hypothesis A after B has been proved.



$\frac{\begin{matrix}[A]\\ \vdots \\ B\end{matrix}}{A \Rightarrow B}$ ⇒-i
$\frac{A \quad A \Rightarrow B}{B}$ ⇒-e
H |= (A⇒B)
Suppose premise is true (⇒i)
J1: {A} can be used as hypothesis to prove B
H U {A} |= B
H |= B
Use implication to reduce goal to new subgoal (⇒e)
H |= (A⇒B)
H |= A

Figure 3: Safety Case Templates for =>-Rules.

In the safety case fragment (see Figure 3), we use a justification to record the use of the hypothesis A, and thus to make sure that the introduced hypotheses are tracked properly.

3.6 Universal Quantifiers

The natural deduction calculus can also be used for proofs in predicate logic. The proof rules 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 arbitrary 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 (see Figure 4) record this replacement as justification. The hypotheses available for the subgoals in the ∀-rules are the same as those in the original goals.

$$\frac{\begin{array}{c}[t_x \text{ fresh}]\\ \vdots \\ A[t_x/x]\end{array}}{\forall x \cdot A}\ \forall\text{-}i \qquad \frac{\forall x \cdot A}{A[t_x/x]}\ \forall\text{-}e$$

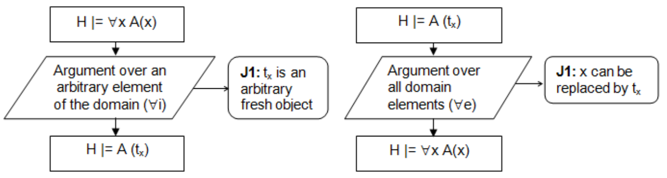

H |= ∀x A(x)
Argument over an arbitrary element of the domain (∀i)
J1: t_x is an arbitrary fresh object
H |= A (t_x)
H |= A (t_x)
Argument over all domain elements (∀e)
J1: x can be replaced by t_x
H |= ∀x A(x)

Figure 4: Safety Case Templates for ∀-Rules.

4 Safety Case Generation Process

To automatically construct the ND proof safety case, we integrate the Muscadet [21] theorem prover with Adelard's ASCE tool [1]. We convert the proofs that are generated by the Muscadet prover into an XML format (i.e., PROOF-XML in Figure 5). The XML file contains all the relevant information that is required for the automatic safety case construction. Subsequently, an XSLT program is used to transform the proofs into another XML (i.e., file GSN-XML in Figure 5) logically representing a safety case, by applying the templates that have been defined in Section 3. The file format was designed so that the derived safety cases can be easily be adapted to different tools or applications. Finally, to present the resulting safety case graphically, we use a Java program to layout the logical information which involved some mathematical calculations in positioning the argument and to convert it into the standard Adelard ASCE file format. Figure 5 summarizes the safety case generation process.

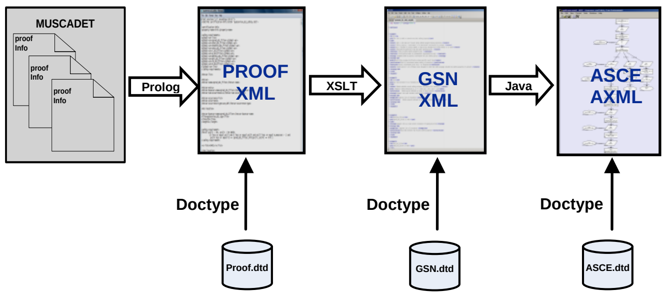


Figure 5: Safety Case Generation Process.

5 Hypothesis Handling

An automated prover typically treats the domain theory D and the certification assumptions A as premises and tries to derive $\wedge$ (D ∪A) $\wedge$ P => C from an empty set of hypotheses. As the proof tree grows, these premises will be turned into hypotheses, using the =>-introduction rule (see Figure 3). In all other rules, the hypotheses are simply inherited from the goal to the subgoals. However, not all premises will actually be used as hypotheses 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 =>-introduction (see Figure 6). We distinguish between the hypotheses that are actually used in the proof of the conclusion (i.e., $A_1,\ldots,A_k$) and those that are vacuously discharged by the =>-introduction (i.e., $A_{k+1},\ldots,A_n$). 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 [2] under which A => B is only valid if the hypothesis A is actually used.

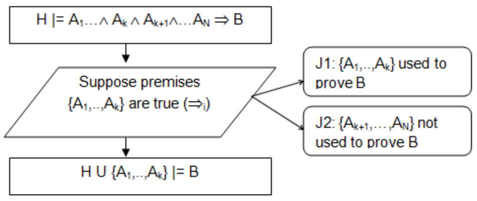

H |= A₁…∧ A_k ∧ A_{k+1}∧…A_N ⇒ B
Suppose premises {A₁,..A_k} are true (⇒ᵢ)
J1: {A₁,..,A_k} used to prove B
J2: {A_{k+1},…,A_N} not used to prove B
H U {A₁,..A_k} |= B

Figure 6: Hypotheses Handling in =>-Introduction Rule.

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 used hypotheses towards the root. By revealing only the 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 7, the use of the hypothesis has_unit(tptp_float_7_0e_minus_1, ang_vel), meaning that a particular floating point variable represents an angular velocity, which has been specified as external assumption, is tracked properly in the safety case, and the validity of its use in deriving the proofs can be checked easily.

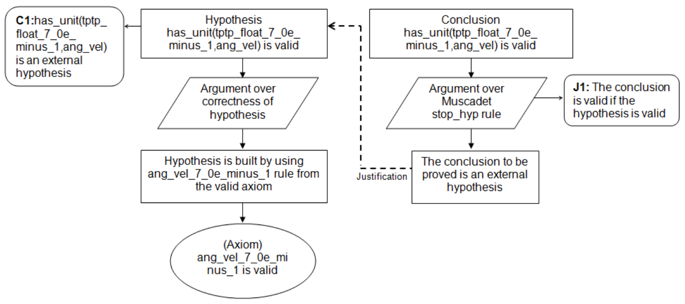

C1:has_unit(tptp_float_7_0e_minus_1,ang_vel) is an external hypothesis
Hypothesis has_unit(tptp_float_7_0e_minus_1,ang_vel) is valid
Conclusion has_unit(tptp_float_7_0e_minus_1,ang_vel) is valid
Argument over correctness of hypothesis
Argument over Muscadet stop_hyp rule
J1: The conclusion is valid if the hypothesis is valid
Hypothesis is built by using ang_vel_7_0e_minus_1 rule from the valid axiom
Justification
The conclusion to be proved is an external hypothesis
(Axiom) ang_vel_7_0e_minus_1 is valid

Figure 7: External Hypothesis.

6 Application to Muscadet

We illustrate our approach by converting proofs created by the AutoCert certification tool [12], which takes a set of requirements, and a domain theory consisting of logical axioms and so-called annotation schemas. These are used to infer logical annotations and construct proof tasks which are sent to the ATP in order to create proofs that the code complies with the requirements.

For these experiments, we used the Muscadet [21] theorem prover during the formal certification of the initialization safety of a component of an attitude control system. Muscadet is based on natural deduction, 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 to the application of the given definitions and lemmas. Figure 8 shows the resulting safety case for a proof found by Muscadet. For some of the "book-keeping" rules (e.g., the elimination of function applications in the elifun-rule in Figure 8) we have not yet defined dedicated safety case templates; these rules are represented by a generic strategy node.

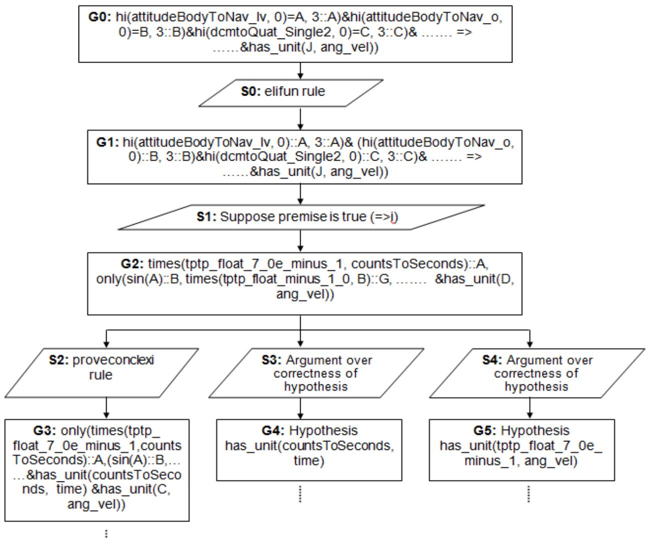

G0: hi(attitudeBodyToNav_lv, 0)=A, 3::A)&hi(attitudeBodyToNav_o, 0)=B, 3::B)&hi(dcmtoQuat_Single2, 0)=C, 3::C)& .... => ......&has_unit(J, ang_vel))
S0: elifun rule
G1: hi(attitudeBodyToNav_lv, 0)::A, 3::A)& (hi(attitudeBodyToNav_o, 0)::B, 3::B)&hi(dcmtoQuat_Single2, 0)::C, 3::C)& .... => ......&has_unit(J, ang_vel))
S1: Suppose premise is true (=>i)
G2: times(tptp_float_7_0e_minus_1, countsToSeconds)::A, only(sin(A)::B, times(tptp_float_minus_1_0, B)::G, ... &has_unit(D, ang_vel))
S2: proveconclexi rule
S3: Argument over correctness of hypothesis
S4: Argument over correctness of hypothesis
G3: only(times(tptp_float_7_0e_minus_1,countsToSeconds)::A,(sin(A)::B,... ...&has_unit(countsToSeconds, time) &has_unit(C, ang_vel))
G4: Hypothesis has_unit(countsToSeconds, time)
G5: Hypothesis has_unit(tptp_float_7_0e_minus_1, ang_vel)

Figure 8: ND Proof Safety Case

7 Proof Abstraction

Directly converting the Muscadet-proofs into safety cases is unfeasible in most practical cases because the proofs contain too many elementary and book-keeping steps. It is thus necessary to abstract the proof. Here, we can apply different approaches. For example, we can remove some of the book-keeping rules (e.g., return_proof) that are not central to the overall argumentation structure. Similarly, we can 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 can 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 9 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.

8 Related Work

Other approaches have been used to address concerns with using proofs for assurance purposes. Many of them try to bring formal proofs into a form closer to human reasoning, to aid with their understanding. Proof visualization tools (e.g., [23]) present the proof in a graphical form, but quickly get overwhelmed by the proof size. Proof verbalization (e.g., [6, 16]) transforms the proofs into natural language but the explanations are often too detailed. Proof abstraction groups multiple low-level steps that represent recurring argumentation patterns into individual abstract steps and thus accentuates the hierarchical structure of the proof [11] but has so far only been applied to interactively constructed proofs. Our work combines abstraction, verbalization and visualization to reveal and present the proof's underlying argumentation structure and top-level assumptions.

Alternatively, proof checkers [20, 24] have been used to increase trust in formal proofs, by demonstrating that every individual step in the proof is correct. However, proof checking does not address the real problem: while errors in the implementations of provers do occur, they are very rare [13]; errors and inconsistencies in the formalization of the domain theory in contrast are much more common, but these are not detected by the standard proof checking techniques.

9 Conclusions

We have described an approach to derive safety cases from software safety proofs found by ATPs. The safety cases serve as a traceable argument that shows validity of the proof. They also highlight and properly track hypotheses that are actually used in deriving the proof, and thus reveal where the proofs depend on top-level assumptions. Greater confidence in the assurance claim can be placed if the rationale behind validity of the proof can be shown. As pointed out by Littlewood *et al.* [19], the probability of a claim, which has been shown by a formal proof, being false, is very low, when the assumptions and evidence are valid. However, there is a non-zero probability that these are not in fact valid. Multi-legged safety cases have been suggested [19] as a technique to compensate for this weakness.

Here, the safety cases provide a "structured reading guide" for the safety proofs and make clear the interactions underlying the proofs. Hence, assurance is no longer implied by the trust in the ATP but follows from an explicitly constructed argument for the proofs. The work we have described here is still in progress. So far, we have automatically derived safety cases for ND proofs found by the Muscadet prover [21]. We are currently working on safety case templates for the remaining inference rules used by Muscadet. We plan to make this technique applicable to generally more powerful resolution provers by converting the resolution proofs to ND style [15]. The straightforward conversion of ND proofs into safety cases is far from satisfactory as they typically contain too

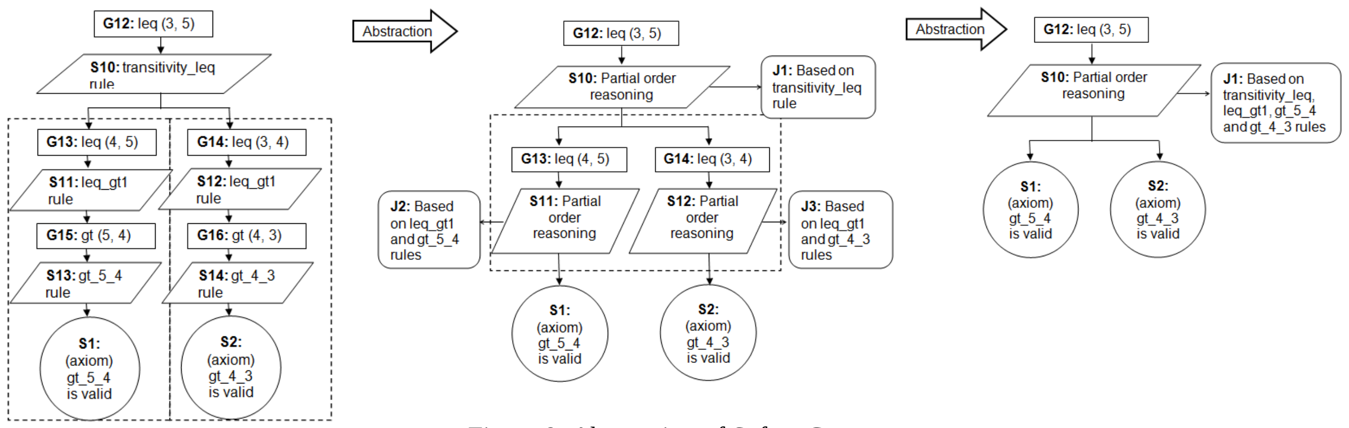

G12: leq (3, 5)
S10: transitivity_leq rule
G13: leq (4, 5)
G14: leq (3, 4)
S11: leq_gt1 rule
S12: leq_gt1 rule
G15: gt (5, 4)
G16: gt (4, 3)
S13: gt_5_4 rule
S14: gt_4_3 rule
S1: (axiom) gt_5_4 is valid
S2: (axiom) gt_4_3 is valid
Abstraction
G12: leq (3, 5)
S10: Partial order reasoning
J1: Based on transitivity_leq rule
G13: leq (4, 5)
G14: leq (3, 4)
J2: Based on leq_gt1 and gt_5_4 rules
S11: Partial order reasoning
S12: Partial order reasoning
J3: Based on leq_gt1 and gt_4_3 rules
S1: (axiom) gt_5_4 is valid
S2: (axiom) gt_4_3 is valid
Abstraction
G12: leq (3, 5)
S10: Partial order reasoning
J1: Based on transitivity_leq, leq_gt1, gt_5_4 and gt_4_3 rules
S1: (axiom) gt_5_4 is valid
S2: (axiom) gt_4_3 is valid

Figure 9: Abstraction of Safety Cases

many details. In practice, a careful use of abstraction is needed [11] and we are working on more abstraction techniques. This work complements our previous work [4] 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 cases as a first step towards a fully-fledged software certificate management system [9] which will provide storage and reporting capabilities for all artifacts. We also believe that the result of our research will be a comprehensive 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 software.

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

References

[1] Adelard ASCE home page, http://www.adelard.com (2007).

[2] A. R. Anderson, N. Belnap. “Entailment: the logic of relevance and necessity”, Princeton University Press (1975).

[3] J. Barnes. “High Integrity Software: The SPARK Approach to Safety and Security”, Addison-Wesley (2003).

[4] N. Basir, E. Denney, B. Fischer. “Constructing a Safety Case for Automatically Generated Code from Formal Program Verification Information”, in: *Proc. SAFECOMP 08,* LNCS 5219, pp. 249-262 (2008).

[5] J.C. Bicarregui, J.S. Fitzgerald, P.A. Lindsay, R. Moore, B. Ritchie. “Proof in VDM: A Practitioner’s Guide”, Springer (1993).

[6] D. Chester. “The Translation of Formal Proofs into English”, *Artificial Intelligence* 7(3), pp. 261-278 (1976).

[7] E.M. Clarke, J.M. Wing. “Formal methods: state of the art and future directions”, *ACM Computing Surveys* 28(4), pp. 626-643 (1996).

[8] E. Denney, B. Fischer. “Correctness of source-level safety policies”, in: *Proc. FM 2003: Formal Methods*, LNCS 2805, pp. 894–913 (2003).

[9] E. Denney, B. Fischer. “Software certification and software certificate management systems (Position paper)”, in: *Proc. ASE Workshop on Software Certificate Management Systems*, ACM, pp.1-5 (2005).

[10] E. Denney, B. Fischer. “A verification-driven approach to traceability and documentation for auto-generated mathe matical software.” in: *Proc. ASE '09*, (to appear 2009).

[11] E. Denney, J. Power, K. Tourlas. “Hiproofs: A hierarchical notion of proof tree”, *ENTCS*, 155, pp.341-359 (2006).

[12] E. Denney, S. Trac, "A Software Safety Certification Tool for Automatically Generated Guidance, Navigation and Control Code", in: *Proc. IEEE Aerospace Conference Big Sky, MT*, (2008).

[13] M. Garnacho, M. Prin. “Convincing Proofs for Program Certification”, in: *Proc. Intl. Workshop SafeCert, ENTCS 17693*, pp. 41-56 (2008).

[14] G. Gentzen. “Investigations into Logical Deductions”, The Collected Papers of Gerhard Gentzen, M.E.Szabo (ed), North-Holland, Amsterdam. pp. 68-131 (1969).

[15] X. Huang. “Translating Machine-Generated Resolution Proofs into ND-Proofs at the Assertion Level”, in: *Proc. 4th Pacific Rim Intl. Conference on Artificial Intelligence*, LNCS 1114, pp. 399-410 (1996).

[16] X. Huang, A. Fiedler. “Proof verbalization as an application of NLG”, in: *Proc 15th Intl. Joint Conference on Artificial Intelligence*, Morgan Kaufmann, pp. 965–970 (1997).

[17] M. Huth, M. Ryan. “Logic in Computer Science Modelling and Reasoning about Systems”, 2nd Edition, Cambridge University Press (2004).

[18] T. P. Kelly. “Arguing safety a systematic approach to managing safety cases”, PhD Thesis, University of York (1998).

[19] B. Littlewood, D. Wright. “The Use of Multilegged Argu ments to Increase Confidence in Safety Claims for Software-Based Systems: A Study Based on a BBN Analysis of an Ide alized Example”, *IEEE Trans. Software Eng*. 33(5), pp. 347-365 (2007).

[20] W. McCune, O. Shumsky-Matlin. “Ivy: A Preprocessor and Proof Checker for First-Order Logic”, *Computer-Aided Reasoning: ACL2 Case Studies*, Advances in Formal Methods (4), Kluwer Academic Publishers, pp. 265–282 (2000).

[21] D. Pastre. “MUSCADET 2.3: A Knowledge-Based Theorem Prover Based on Natural Deduction”, in: *Proc Intl. Joint Conference on Automated Reasoning*, LNCS 2083, pp. 685-689 (2001).

[22] PolySpace Technologies, http://www.polyspace.com, 2007.

[23] S. Trac, Y. Puzis, G. Sutcliffe. “An interactive derivation viewer”, in: *Proc. Intl. Workshop User Interfaces for Theorem Provers*, ENTCS 174, Elsevier, pp.109-123 (2007).

[24] G. Sutcliffe, D. Belfiore. “Semantic Derivation Verification”, in: *Proc. 18th Florida Artificial Intelligence Research Symposium*, pp.641–646 (2005).