Assurance Cases for Proofs as Evidence

Sagar Chaki   Arie Gurfinkel   Kurt Wallnau   Charles Weinstock
Software Engineering Institute, Carnegie Mellon University
Pittsburgh, PA
{chaki|arie|kcw|weinstock}@sei.cmu.edu

Abstract

Proof-carrying code (PCC) provides a "gold standard" for establishing formal and objective confidence in program behavior. However, in order to extend the benefits of PCC – and other formal certification techniques – to realistic systems, we must establish the correspondence of a mathematical proof of a program's semantics and its actual behavior. In this paper, we argue that assurance cases are an effective means of establishing such a correspondence. To this end, we present an assurance case pattern for arguing that a proof is free from various proof hazards. We also instantiate this pattern for a proof-based mechanism to provide evidence about a generic medical device software.

1 Introduction

Today's information-based society is dependent on software for its well-being. Software is ubiquitous and invisible in everything from entertainment to critical infrastructure; "out of sight, out of mind" describes current public sentiment about this dependence. Moreover, software components are being interconnected in ways that were never anticipated, or in some cases intended. The adverse effects of software failures resulting from this increased coupling are difficult to contain. Software development has become a commodity service, and software supply chains span the globe; the provenance of any complex software package is, and will likely remain, unknown. Thus, there is an urgent and well recognized need for justifiable confidence that software will behave as intended by the consumer. Moreover, the source of this confidence must be the software artifact itself, and not the identity of, or the processes used by, the software producer. Known provenance and processes are useful, but are not always available to consumers, and do not guarantee acceptable behavior.

Proof-carrying code (PCC) [9] is a "gold standard" for establishing justifiable confidence in program behavior, and has been the epicenter of many recent technical advancements. For example, Chaki et al. have developed [3] a certifying model checker (CMC) and associated machinery to produce PCC against any linear temporal logic (LTL) specification. However, in order to extend the benefits of PCC, and other formal technologies, to large complex systems, we must establish correspondence of a mathematical proof within a *formal system* and the behavior that is exhibited in the *real world*. In this paper, we argue that assurance cases [5] (or cases, for short) provide an effective solution to this correspondence problem. An assurance case is a structured argument that a claimed system-level property has been achieved. Assurance cases employ defeasible reasoning, where a premise (ultimately, evidence) *usually* implies a conclusion. Defeasible reasoning offers an intermediate ground between formal notions of soundness and completeness and the intrinsic uncertainty and incompleteness of any large scale, complex system.

We present an assurance case pattern for arguing that any formal proof is free from various hazards to proof validity. Our pattern handles proof hazards arising from the *use* of the formal technology (did we model the right behavior?), as well as from the technology *itself* (do we trust the theorem prover?). Our approach has several benefits. First, it captures, in pattern form, a variety of threats to the validity of any formal evidence, in effect normalizing and improving the quality of such evidence. Second, the pattern can be extended to argue about the benefits of specific technologies, for example to show why PCC allows us to eliminate model checkers, theorem provers, and even compilers from the trusted computing base. Finally, case patterns and their instances are amenable to being expressed in precise notation, recorded, shared, reviewed, and revised. We demonstrate the effectiveness of our case pattern by instantiating it for a specific application of CMC and PCC technology to provide evidence about software in a hypothetical infusion pump. Our results are preliminary, but encouraging. We believe that, ultimately, such use of cases improves the transitionability of formal techniques to practical situations.

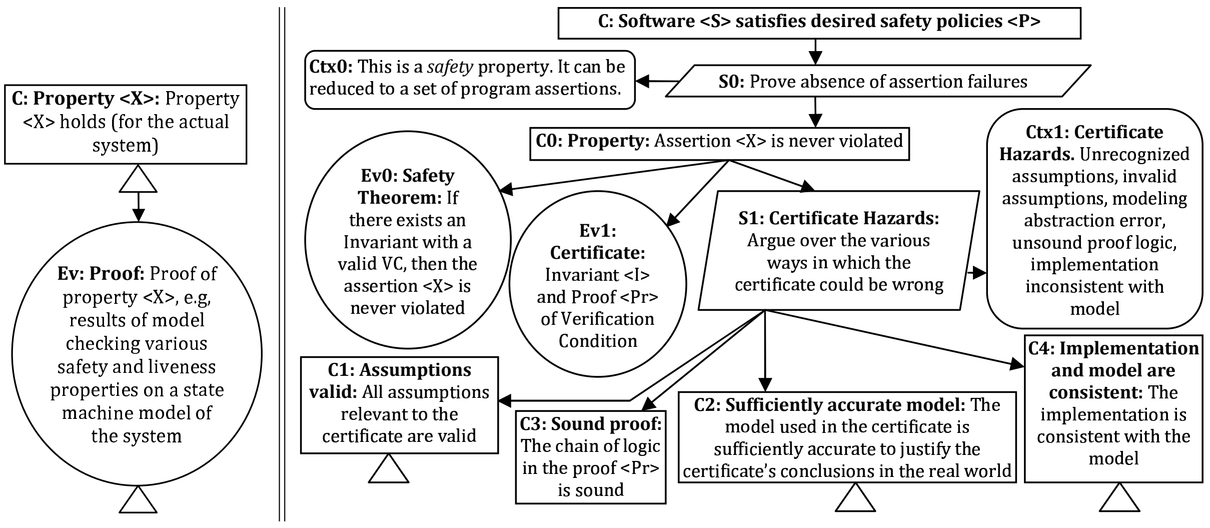
C: Property <X>: Property <X> holds (for the actual system)
Ev: Proof: Proof of property <X>, e.g, results of model checking various safety and liveness properties on a state machine model of the system
C: Software <S> satisfies desired safety policies <P>
Ctx0: This is a safety property. It can be reduced to a set of program assertions.
S0: Prove absence of assertion failures
C0: Property: Assertion <X> is never violated
Ev0: Safety Theorem: If there exists an Invariant with a valid VC, then the assertion <X> is never violated
Ev1: Certificate: Invariant <I> and Proof <Pr> of Verification Condition
S1: Certificate Hazards: Argue over the various ways in which the certificate could be wrong
Ctx1: Certificate Hazards. Unrecognized assumptions, invalid assumptions, modeling abstraction error, unsound proof logic, implementation inconsistent with model
C1: Assumptions valid: All assumptions relevant to the certificate are valid
C3: Sound proof: The chain of logic in the proof <Pr> is sound
C2: Sufficiently accurate model: The model used in the certificate is sufficiently accurate to justify the certificate's conclusions in the real world
C4: Implementation and model are consistent: The implementation is consistent with the model

Figure 1: (Left) GSN notation; (Right) top-level GIP assurance case pattern.

2 Assurance Cases and Infusion Pump Scenario

An assurance case uses a claims-argument-evidence structure to demonstrate the truth of some assertion. It consists of a top-level claim supported by subclaims. Each subclaim is further decomposed into sub-subclaims, and so on, until a claim is directly supported by evidence, i.e., data that is sufficient to support a claim without further argument. Typical examples of evidence are test results, analyses, information about the competency of personnel, etc. The quality of the case (i.e., its soundness and the extent to which it is convincing in supporting its top-level claim) depends on the claim structure and the quality of the presented evidence.

An assurance case is an example of defeasible reasoning, i.e., reasoning where "the corresponding argument is rationally compelling but not deductively valid ... the relationship of support between premises and conclusion is a tentative one, potentially defeated by additional information" [10]. The logical form of a defeasible inference is: $\textbf{if}\ E\ \textbf{then}(\textbf{usually})\ C\ \textbf{unless}\ R,S,T,\ \textbf{etc}$. In other words, claim C follows from evidence E, unless this inference is invalidated by deficiencies R, S, T, etc. The set of deficiencies is never completely known. Even if we argue $\neg R$, $\neg S$, and $\neg T$, new information (e.g., U) could invalidate the $E \Rightarrow C$ inference, or the demonstration of, say, $\neg R$. Therefore, confidence in C is improved by capturing as many deficiencies as possible, and showing their absence.

Infusion Pump Scenario. An infusion pump infuses fluids, medication or nutrients into a patient's circulatory system. Our case study involves a Generalized Infusion Pump (GIP), which includes a built-in drug library. The drug library contains a list of drugs, and, for each drug, the following: (a) drug name, (b) drug concentration, and (c) for each clinical setting, the soft (and hard) minimum (and maximum) allowed infusion rates. The acceptable infusion rate in an emergency environment may be significantly higher than that in a patient room. The acceptable infusion rate for an adult may be significantly higher than for an infant. The GIP consults the drug library when the caregiver is programming an infusion.

We assume the following scenario: (i) the GIP uses an established software and hardware architecture, (ii) the GIP software is supplied by third parties, and (iii) the GIP manufacturer requires certifiable assurance that the delivered GIP software satisfies the following three (publicly specified) safety policies: **(P1)** if the infusion rate of the selected drug is within the soft bounds appropriate to the setting, the GIP accepts the programming; **(P2)** if the infusion rate is outside of the soft bounds but within the hard bounds the GIP accepts the programming only after a warning and a required override by the caregiver;

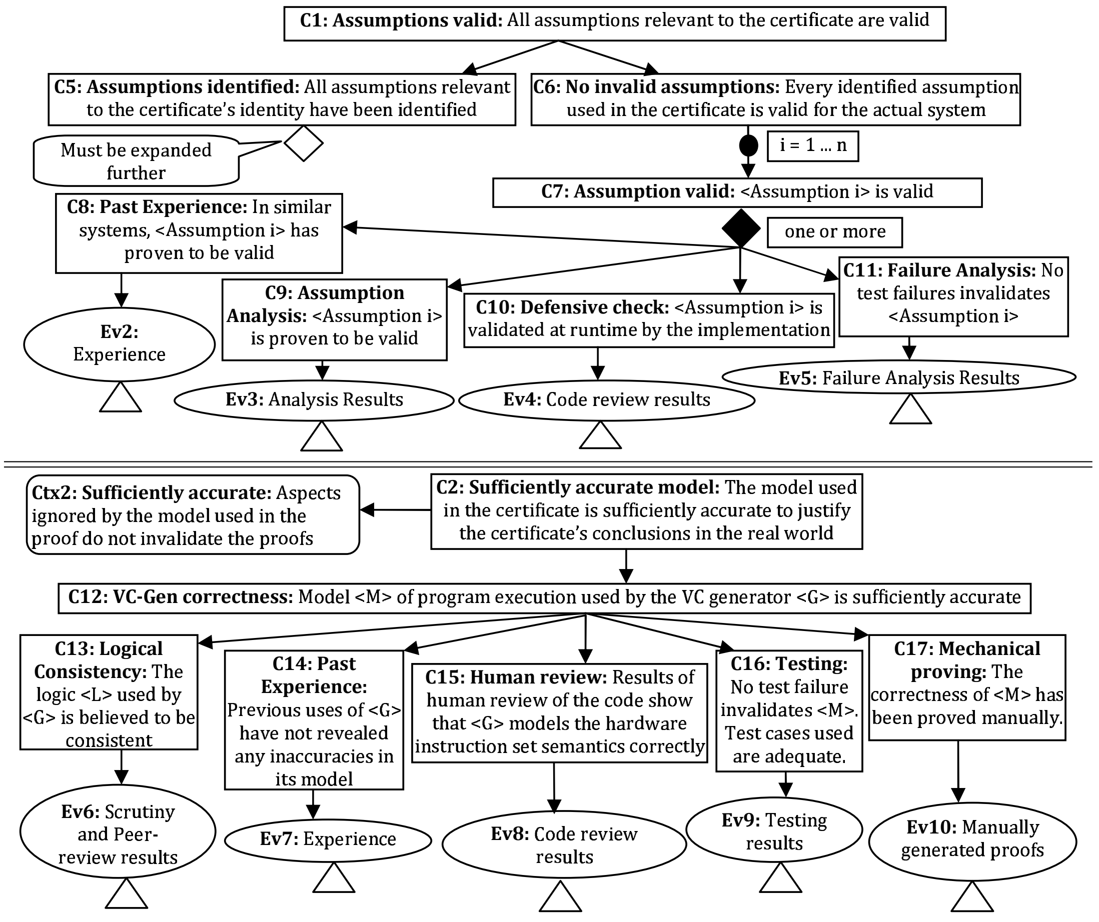
C1: Assumptions valid: All assumptions relevant to the certificate are valid
C5: Assumptions identified: All assumptions relevant to the certificate's identity have been identified
C6: No invalid assumptions: Every identified assumption used in the certificate is valid for the actual system
Must be expanded further
i = 1 ... n
C7: Assumption valid: <Assumption i> is valid
C8: Past Experience: In similar systems, <Assumption i> has proven to be valid
one or more
C9: Assumption Analysis: <Assumption i> is proven to be valid
C10: Defensive check: <Assumption i> is validated at runtime by the implementation
C11: Failure Analysis: No test failures invalidates <Assumption i>
Ev2: Experience
Ev3: Analysis Results
Ev4: Code review results
Ev5: Failure Analysis Results
Ctx2: Sufficiently accurate: Aspects ignored by the model used in the proof do not invalidate the proofs
C2: Sufficiently accurate model: The model used in the certificate is sufficiently accurate to justify the certificate's conclusions in the real world
C12: VC-Gen correctness: Model <M> of program execution used by the VC generator <G> is sufficiently accurate
C13: Logical Consistency: The logic <L> used by <G> is believed to be consistent
C14: Past Experience: Previous uses of <G> have not revealed any inaccuracies in its model
C15: Human review: Results of human review of the code show that <G> models the hardware instruction set semantics correctly
C16: Testing: No test failure invalidates <M>. Test cases used are adequate.
C17: Mechanical proving: The correctness of <M> has been proved manually.
Ev6: Scrutiny and Peer-review results
Ev7: Experience
Ev8: Code review results
Ev9: Testing results
Ev10: Manually generated proofs

Figure 2: Case patterns for "assumptions valid" (top) and "sufficiently accurate model" (bottom).

(P3) the GIP cannot be programmed with an infusion rate outside of the hard bounds.

3 GIP Assurance Case Pattern and Instantiation

We use the graphical goal structuring notation (GSN) [5] to express assurance cases. Fig. 1(left) shows, in GSN, the case that "property `<X>`" holds because there is a proof of the property. Specifically, "property `<X>` holds" is the claim, and "Proof of property `<X>`" is the evidence presented in support of this claim. A rectangle indicates a claim, always phrased as a predicate. A circle (or ellipse) indicates evidence (always stated in a noun phrase), and the arrow linking the claim to the evidence implies that the claim is supported by the evidence. The little triangles at the bottom of the rectangle and circle indicate that the claim and evidence are generic and need to be instantiated when this pattern is applied. Angled brackets (`<>`) characterize what is to be instantiated. In the remaining cases, we omit such triangles when there is an explicit `<X>` to be instantiated. Also, we use the following additional GSN features. A parallelogram refers to a strategy, while a rounded rectangle refers to a context. Empty diamonds refer to parts that have been left out, but must be expanded further. Solid diamonds refer to a choice between various alternatives. A solid circle denotes iteration.

Fig. 1(right) shows, in GSN, the top-level assurance case pattern for the generic claim "Software `<S>` satisfies desired safety policies `<P>`". It leaves the following four sub-claims to be expanded further:

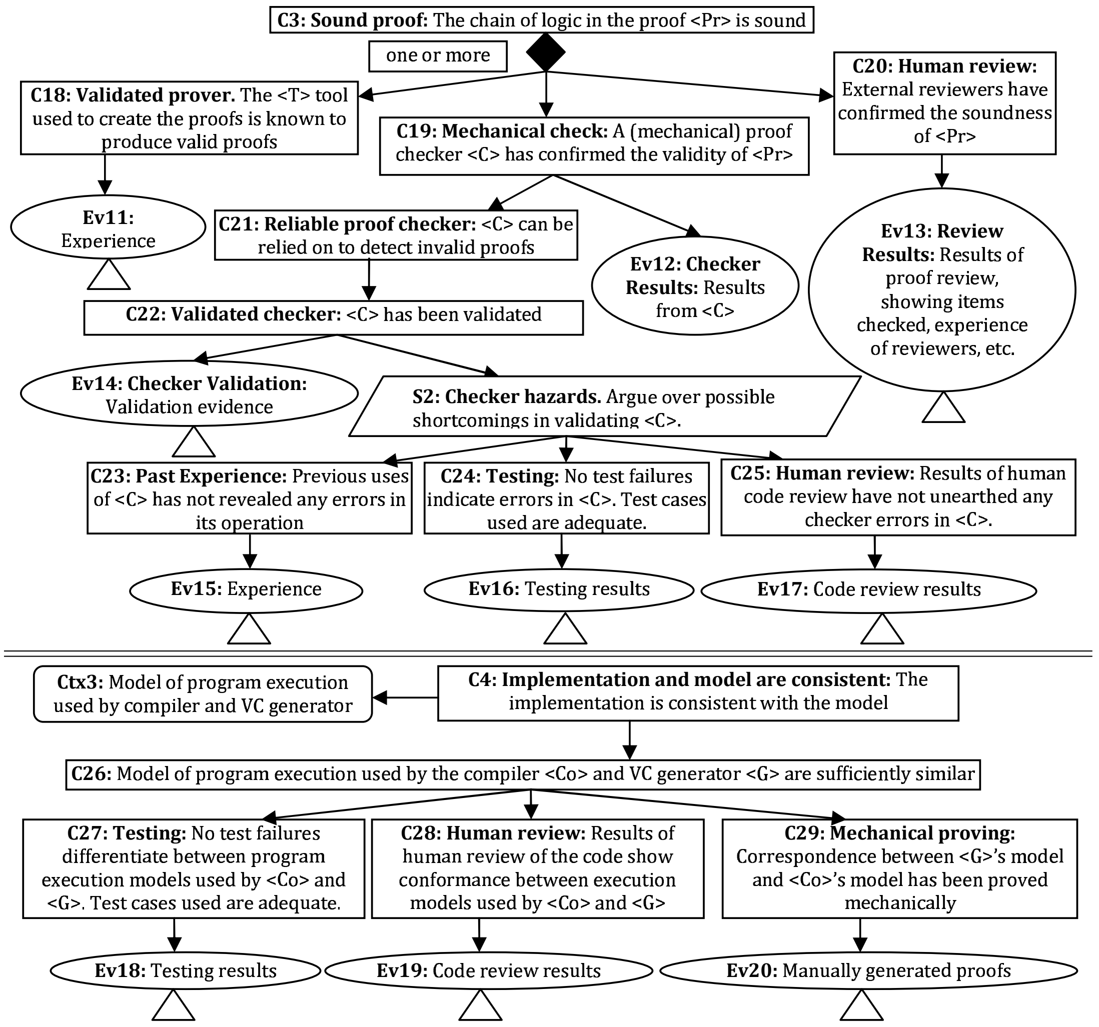
C3: Sound proof: The chain of logic in the proof <Pr> is sound
one or more
C18: Validated prover. The <T> tool used to create the proofs is known to produce valid proofs
C19: Mechanical check: A (mechanical) proof checker <C> has confirmed the validity of <Pr>
C20: Human review: External reviewers have confirmed the soundness of <Pr>
Ev11: Experience
C21: Reliable proof checker: <C> can be relied on to detect invalid proofs
Ev12: Checker Results: Results from <C>
Ev13: Review Results: Results of proof review, showing items checked, experience of reviewers, etc.
C22: Validated checker: <C> has been validated
Ev14: Checker Validation: Validation evidence
S2: Checker hazards. Argue over possible shortcomings in validating <C>.
C23: Past Experience: Previous uses of <C> has not revealed any errors in its operation
C24: Testing: No test failures indicate errors in <C>. Test cases used are adequate.
C25: Human review: Results of human code review have not unearthed any checker errors in <C>.
Ev15: Experience
Ev16: Testing results
Ev17: Code review results
Ctx3: Model of program execution used by compiler and VC generator
C4: Implementation and model are consistent: The implementation is consistent with the model
C26: Model of program execution used by the compiler <Co> and VC generator <G> are sufficiently similar
C27: Testing: No test failures differentiate between program execution models used by <Co> and <G>. Test cases used are adequate.
C28: Human review: Results of human review of the code show conformance between execution models used by <Co> and <G>
C29: Mechanical proving: Correspondence between <G>'s model and <Co>'s model has been proved mechanically
Ev18: Testing results
Ev19: Code review results
Ev20: Manually generated proofs

Figure 3: Case patterns for "sound proof" (top) and "implementation and model are consistent" (bottom).

(C1) assumptions valid, **(C2)** sufficiently accurate model, **(C3)** sound proof, and **(C4)** implementation and model are consistent. The case pattern for **(C1)** and **(C2)** are shown in Fig. 2. Note that the case for **(C1)** has a sub-claim "assumptions identified" that we do not expand further for brevity. The case patterns for **(C3)** and **(C4)** are shown in Fig. 3.

Certification Mechanism. We consider a specific certification mechanism, called PCCCMC, that uses a combination of PCC and CMC to provide formal evidence of safe runtime behavior of programs [3]. The input to PCCCMC is a C program *P* containing an assertion ASRT. The output is a proof-certificate consisting of an invariant INVAR and a proof PROOF. Let *P* be the GIP software such that ASRT enforces the desired safety policies **P1–P3**. Then a run of PCCCMC on *P* consists of the following steps: (i) INVAR is generated using a certifying software model checker CMC; (ii) a verification condition VC is generated using weakest preconditions by a VCGEN tool; intuitively, VC is a logical formula in a suitable logic $\mathscr{L}$ expressing that INVAR is inductive and implies ASRT; (iii) PROOF is generated by checking the validity of VC using a proof-generating theorem prover PROVER, (iv) PROOF is checked via a CHECKER. The correctness of PCCCMC relies on the "safety theorem" which basically states that *P* does not violate ASRT at runtime if there exists an INVAR for which the VC is valid.

Pattern Instantiation. We now instantiate our assurance case patterns in the context of PCCCMC. In the top-level pattern (see Fig. 1) we instantiate `S` with the GIP Software, `P` with **P1–P3**, and `X` with ASRT. Also, we instantiate `I` with INVAR, and `P` by PROOF. In the pattern for **C1**, we identify and instantiate as many assumptions as possible that are relevant to the certificate. In the pattern for **C2**, we instantiate `G` by VCGEN, `M` by the execution semantics of the GIP Software used by VCGEN, and `L` by $\mathscr{L}$. In the pattern for **C3**, we instantiate `P` by PROOF, `T` by PROVER, and `C` by CHECKER. Finally, in the pattern for **C4**, we instantiate `G` by VCGEN and `C` by COMPILER used to compile the GIP software before deployment.

Related Work. Kelly [5] provides more information on assurance cases and GSN. Weaver [11] documents the use of assurance cases (and case patterns) in software. Assurance cases have been used to address system safety [6], and to justify safety and dependability claims [7]. Arney et al. have developed a set of requirements and a hazard analysis for a generic infusion pump [1]. Goodenough and Weinstock [4] explore demonstrating the quality of the evidence in an assurance case, and using assurance cases for medical devices [12]. Basir et al. [2] have looked at automatically generating safety cases from the formal annotations used to construct Hoare-style proofs of program correctness. Our approach is less automated, but potentially applicable to a wider class of proof-generation techniques. PCC [9] was introduced by Necula and Lee and provides an effective means for providing objective evidence of memory safety properties of low-level. CMC [8] aims to generate proof-certificates by extending model checking algorithms. Chaki et al. [3] have explored combinations of PCC and CMC to generate proof-certificates of expressive properties on low-level programs. Our work is aimed at extending these, and other, formal techniques to provide objective confidence about the safe execution of realistic systems.

Conclusion and Future Work. We report on preliminary work in using assurance cases to bridge the gap between a proof about a program's semantics in a formal system, and its actual behavior in the real world. To this end, we present an assurance case pattern for arguing that a proof is free from various validity hazards. We also instantiate this pattern for a specific application of formal certification technology to an infusion pump software. An important question is if our pattern is instantiable with formal certification schemes other than PCCCMC, and how to make it more robust and complete.

References

[1] D. Arney, R. Jetley, P. Jones, I. Lee, and O. Sokolsky. Generic Infusion Pump Hazard Analysis and Safety Requirements. Technical report MS-CIS-08-31, University of Pennsylvania, October 2008.

[2] N. Basir, E. Denney, and B. Fischer. Constructing a safety case for automatically generated code from formal program verification information. In *Proc. of SAFECOMP*, 2008.

[3] S. Chaki, J. Ivers, P. Lee, K. Wallnau, and N. Zeilberger. Model-driven construction of certified binaries. In *Proc. of MODELS*, 2007.

[4] J. Goodenough and C. Weinstock. Hazards to Evidence: Demonstrating the Quality of Evidence in an Assurance Case. Technical Report CMU/SEI-2008-TN-016, SEI, 2008. in preparation.

[5] T. Kelly. *Arguing Safety*. PhD thesis, Univ. of York, 1998.

[6] T. Kelly and R. Weaver. The Goal Structuring Notation – A Safety Argument Notation. In *Proc. of the Dependable Systems and Networks Workshop on Assurance Cases*, 2004.

[7] L. Millett. Software for Dependable Systems: Sufficient Evidence?, 2007. http://www.nap.edu/catalog.php?record_id=11923.

[8] K. S. Namjoshi. Certifying Model Checkers. In *Proc. of CAV*, 2001.

[9] G. C. Necula. Proof-Carrying Code. In *Proc. of POPL*, 1997.

[10] Stanford Encyclopedia of Philosophy: Defeasible Reasoning, 2005.

[11] R. Weaver. *The Safety of Software – Constructing and Assuring Arguments*. PhD thesis, Univ. of York, 2003.

[12] C. Weinstock and J. Goodenough. Towards Assurance Cases for Medical Devices. Technical Report CMU/SEI-2009-TN-018, SEI, 2009. in preparation.