Toward a Dependability Case Language and Workflow for a Radiation Therapy System

Michael D. Ernst[1], Dan Grossman[1], Jon Jacky[2], Calvin Loncaric[1], Stuart Pernsteiner[1], Zachary Tatlock[1], Emina Torlak[1], and Xi Wang[1]

1 **Computer Science and Engineering, University of Washington**
Seattle, Washington USA
`{mernst, djg, loncaric, spernste, ztatlock, emina} @ cs.washington.edu`

2 **Radiation Oncology, University of Washington**
Seattle, Washington USA
`jon@washington.edu`

Abstract

We present a near-future research agenda for bringing a suite of modern programming-languages verification tools—specifically interactive theorem proving, solver-aided languages, and formally defined domain-specific languages—to the development of a specific safety-critical system, a radiotherapy medical device. We sketch how we believe recent programming-languages research advances can merge with existing best practices for safety-critical systems to increase system assurance and developer productivity. We motivate hypotheses central to our agenda: That we should start with a single specific system and that we need to integrate a variety of complementary verification and synthesis tools into system development.

1998 ACM Subject Classification D.2.4 Software/Program Verification; J.3 Life and Medical Sciences

Keywords and phrases Synthesis, Proof Assistants, Verification, Dependability Cases, Domain Specific Languages, Radiation Therapy

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Safety-critical systems—containing software in which errors can lead to death, injury, and wide-scale destruction—are nothing new. Experts have built them for decades at great expense and, despite well-known failures, our trust in software continues to grow as we increasingly depend on medical devices, transportation networks, financial exchanges, and other critical infrastructure where software plays a critical role.

In recent years, only a tiny fraction of mainstream programming-languages research has *directly* targeted safety-critical systems.[1] This is surprising given the astonishing advances in automatic verification and synthesis that many of us surely *believe* could, with appropriate adaptations and focus, reduce costs and improve reliability for safety-critical systems. After all, we can now build substantial formally verified software infrastructure like the CompCert compiler [30], a reference monitor for a modern web-browser [24], a full operating

[1] Note that there are a handful of significant exceptions, such as the ASTREE analyzer in avionics [3], Galois' work on highly dependable Haskell platforms [12], and Praxis' work on safety-critical systems built in Spark/Ada [1].

Conference title on which this volume is based on.
Editors: Billy Editor and Bill Editors; pp. 1–11
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

system kernel [27], and cryptographic protocols [2]. However, note that the majority of these systems are traditional, core computing infrastructure. It is not surprising that the community has focused on software systems closer to our deepest experience, much as there are disproportionately many plays about the theatre and novels about novelists.

But, as we discuss in this paper, there are important safety-critical systems that are architected, specified, developed, and maintained in fundamentally different ways. Successfully bringing cutting-edge programming-languages verification techniques into modern safety-critical systems will require a long-term research agenda and deep collaborations with domain experts. We are beginning such an agenda, and this paper lays out our initial plans and hypotheses. It is a call for others to pursue similar or complementary approaches. It is a preview against which we can judge progress in a few years.

We have several high-level hypotheses, each expanded upon in the rest of this paper:

- *Our methodology should proceed from the specific to the general*, working first on one particular safety-critical system, then a second and a third, and [only] then trying to abstract to general principles. We are focusing on helping develop the third-generation of the Clinical Neutron Therapy System (Section 2) at the University of Washington Medical Center, a sophisticated medical device used for about 30 years without incident on our campus.
- *The right model for building safety-critical systems is Jackson's approach of dependability cases* (see Section 3), in which heterogeneous evidence of reliability is brought together in an explicit, layered way to connect system requirements to low-level implementation details. Our goal is not to replace the human element in this process, but to enrich it with formal and automatic verification of key pieces. To this end, we plan to develop a high-level *dependability case language* (DCL) for specifying, checking, and evolving dependability cases.
- *No one verification technique is right for the entire dependability case.* In particular, we hope to use an integrated workflow of complementary technologies (Section 3) that are rarely used together on the same project today, namely:

  - A formally verified domain-specific language (DSL) for writing the safety-critical software. Indeed, the strict requirements of such systems make DSLs (or highly restricted subsets of more general languages) the standard approach already.
  - Coq [5] for proving key semantic properties of the DSL implementation, and key correctness properties of shared libraries and components. Infrastructure bugs are *contagious* in the sense that a bug in the infrastructure may cause faults in any application code running on top of it. Furthermore, infrastructure code changes rarely. It therefore makes sense to apply heavyweight verification to language implementations, as evidenced [37, 29] by the reliability of verified language platforms such as CompCert [30] and Bedrock [4].
  - Solver-aided verification and synthesis for accelerating code reviews and revisions of DSL applications. Even safety-critical applications change frequently enough to make heavyweight verification (e.g., with Coq) prohibitive. At this level, the role of tools is to accelerate the standard development process. While static analysis is routinely used in this way [14], solver-aided tools are not, despite their successful application within many DSLs (see, e.g., [35, 36]). These tools can reason about complex program properties that are often needed to establish end-to-end dependability—and that are poorly supported by static analyzers.
  - Alloy [16] for describing and checking the system architecture and design. A design-level tool must enable fast iteration and prototyping. As such, the tool must be interactive

Figure 1 The CNTS console, collimator, and patient setup in gantry.

and capable of producing counterexamples. It must also support a rich logic for partial formalization of critical aspects of the system—not all parts of a design need to be subjected to formal analysis, and those that do not should be easy to abstract. Alloy makes it easy both to model and check designs, and it has a long history of discovering flaws in designs of safety- and correctness-critical systems (e.g., [10, 32, 25, 38]).

2 The Clinical Neutron Therapy System (CNTS)

The Clinical Neutron Therapy System (CNTS) at the University of Washington Medical Center (UWMC) is an advanced radiotherapy installation for treating tumors with neutron radiation. Neutron therapy is highly effective at treating inoperable tumors that are resistant to conventional electron- or photon-based radiation therapy. But the equipment for neutron therapy is also an order of magnitude more expensive than conventional radiation therapy. For this reason, CNTS is one of only three neutron therapy installations in the United States, and thus it depends extensively on custom software developed by the CNTS staff. Through deep expertise and great care, the CNTS staff has achieved a remarkable safety record, with no major incidents in over 30 years of service treating patients.

2.1 The CNTS Installation

The system consists of a cyclotron and a treatment room (Figure 1) with a computer-operated leaf collimator. The cyclotron generates a broad beam of neutrons that passes through the collimator on its way to the patient. The collimator consists of forty steel leaves and several filters, which control the shape and intensity of the beam, respectively. The collimator is mounted on a gantry that rotates 360 degrees so that the beam can enter the patient from any angle. The entry point is additionally controlled through the position of the couch (with five degrees of motion freedom) on which the patient lies during treatment.

A treatment beam prescription specifies the positions of all moving components (couch, leaves, and filters), the daily radiation dose, and the total radiation dose. Radiation therapy technologists manually ensure that all components are positioned correctly before beginning treatment. The therapy control system ensures that *the neutron beam can turn on and remain on only when all moving components are set as prescribed, and when the daily and total dose are less than prescribed.* This is the primary safety requirement for CNTS.

The therapy control system is carefully designed to enforce the CNTS safety requirement [19, 23, 20]. For example, a critical aspect of enforcing the requirement is to ensure that the system can always reach a safe state in which the beam is turned off. The design of the

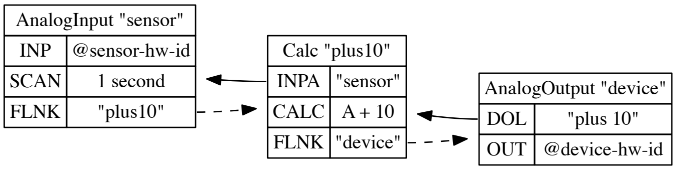


Figure 2 An example program in EPICS_0. Dashed lines indicate control transfers while solid lines indicate data dependencies.

system therefore provides a non-software path from any state to a safe state. In particular, all basic therapy controls, such as turning the beam on and off, are implemented with hardware relays, programmable logic controllers (PLCs), or embedded microcomputers with programs in read-only memory. The software controller, which runs on general-purpose computers, is used only for advanced therapy control functions.[2] As a result, CNTS can reach a safe state (by turning off the beam through a basic control) even if its software controller crashes.

While bugs in the software controller cannot prevent the beam from being turned off, they could still cause a violation of the CNTS safety requirement, with deadly consequences. For example, the software controller is responsible for loading prescriptions from the patient database into the low-level device controllers. If this is done incorrectly, a patient could receive too much radiation. The software controller is therefore considered to be a safety-critical part of the overall system, and its development has followed rigorous practices and standards [19, 23, 20].

2.2 The CNTS Software Controller

The CNTS software controller has undergone two complete rewrites since 1984. The vendor-provided controller was originally written in FORTRAN; in 1999 it was replaced with a custom C program [19, 23]. The latter is now being phased out in favor of a new controller [20] written in a subset of a general-purpose dataflow language from the Experimental Physics and Industrial Control System (EPICS) [11]. This subset, which we call EPICS_0, forms a tiny embedded DSL that consists of just 19 EPICS constructs.

Figure 2 shows a simple program in EPICS_0 that reads sensor input once per second, adds 10 to the most recently read value, and writes the resulting value to some output device. EPICS_0 is like EPICS in that the programmer must explicitly specify both control transfers and data dependencies. For example, programmers need to specify both that after the “sensor” node finishes processing, it adds the “plus10” node to the processing queue (indicated by the dashed arrow starting from the “`FLNK`” field) and also that the “plus10” node will read the most recent value from the “sensor” node into its `INPA` field (indicated by the solid arrow starting from the “`INPA`” field). However, unlike EPICS, EPICS_0 programs guarantee that these links are always well-formed (e.g., links never point to non-existent nodes and are loop free).

[2] These include retrieving prescriptions from the patient database, loading prescribed settings into low-level device controllers, comparing the prescribed settings to those read back from the controllers, instructing the controllers in a particular sequence, and storing a record of each treatment in the database [20].

As EPICS is already a popular and time-tested framework for controlling scientific instruments, the advantage of transitioning to EPICS$_0$ is that it provides numerous features that will enable a broader range of therapies at CNTS and thus increase the utility of the system. However, EPICS was not originally developed for clinical use, but rather for particle physics experiments. Adapting such research software to a safety critical presents several challenges. In particular, EPICS has a massive, complex code base, which is too large for the CNTS staff to fully audit. During adoption, EPICS has exhibited behavior that the CNTS staff were unable to explain, leading them to carefully avoid using certain features. In addition, the language has no formal semantics, making it difficult to reason about program's behavior. Finally, it is unclear how to port existing system requirements to this infrastructure in a way that is maintainable for future CNTS staff.

3 A Dependability Case for CNTS: the Language and the Workflow

The CNTS has operated without incident for 30 years, due to the rigorous development [23, 21] and operational [22] standards practiced by the in-house engineering and hospital staff. This level of safety has not been easy to obtain, however, relying on careful manual reasoning, documentation, and extensive testing. Our goal is to aid the CNTS engineers in maintaining the system's impressive safety record—with less effort and with higher confidence—as it transitions to the new EPICS$_0$ software controller. To that end, we propose to construct an explicit dependability case [15] for CNTS, with the help of an integrated workflow consisting of a language for expressing dependability claims and tools for supporting them with evidence.

3.1 The Dependability Case Approach to Building Reliable Systems

Safety and mission-critical systems, like CNTS or the Mars rover [14], are architected, specified, developed, and maintained according to strict best practices, by highly skilled engineers. At the system level, these practices involve detailed requirements, documentation, hazard analysis, and formalization of key parts of the system design. At the code level, they include adherence to stringent coding conventions (see, e.g., [13]), manual code reviews, use of static analysis, and extensive testing. The CNTS engineers, for example, followed [19] these practices when developing the second generation of the CNTS therapy control software—just as the NASA engineers followed them when developing the software for the Mars rover [14]. But is adherence to best practices enough to ensure that the resulting systems are indeed dependable—i.e., that they always satisfy their safety goals and requirements?

Based on a comprehensive two-year study [6] of how dependable software might be built, Jackson [15] argues that best practices alone cannot guarantee dependability. After all, a correctly implemented system may fail catastrophically if its requirements are based on an invalid assumption about its environment.[3] For this reason, Jackson proposes an approach for constructing dependable systems in which engineers produce both a system and an evidence-based argument, or a *case*, that the system satisfies its dependability goals.

In Jackson's approach, a dependability case is a collection of explicitly articulated *claims* that the system (i.e., the software, the hardware, and, if applicable, the human operators) has desired critical *properties.* Each is supported by *evidence* that may take a variety of forms, such as formal proofs, tests suites, and operating procedures. The case as a whole must be *auditable* (by third parties), *complete* (including relevant assumptions about the

[3] Such an assumption was responsible for loss of life in a 1993 landing accident at the Warsaw airport [15].

environment or user behavior), and *sound* (free of false claims and unwarranted assumptions). In essence, it must present a socially consumable proof [9] of the system's dependability.

We believe that the dependability case approach is the right model for developing safety-critical systems. In fact, it is the model that CNTS engineers intuitively followed when constructing the second generation of the therapy control system. As noted in Section 2, the design of the system was explicitly based [23] on the end-to-end safety requirement that the beam may be active only when the machine's settings match the patient's prescription. The ensuing development effort then yielded a collection of artifacts that comprise a rudimentary dependability case: a 200-page document [19] detailing the system requirements, developed in consultation with physicists and clinicians; a 2,100 line Z specification [19] of the therapy control software; a 16,000 LOC implementation of the Z specification in C; a 240-page reference manual [21]; and a 43-page therapist guide [22]. Our plan is to create an explicit case for the latest generation of the therapy control system that is powered by the EPICS$_0$ software controller [20].

3.2 A Dependability Case Language

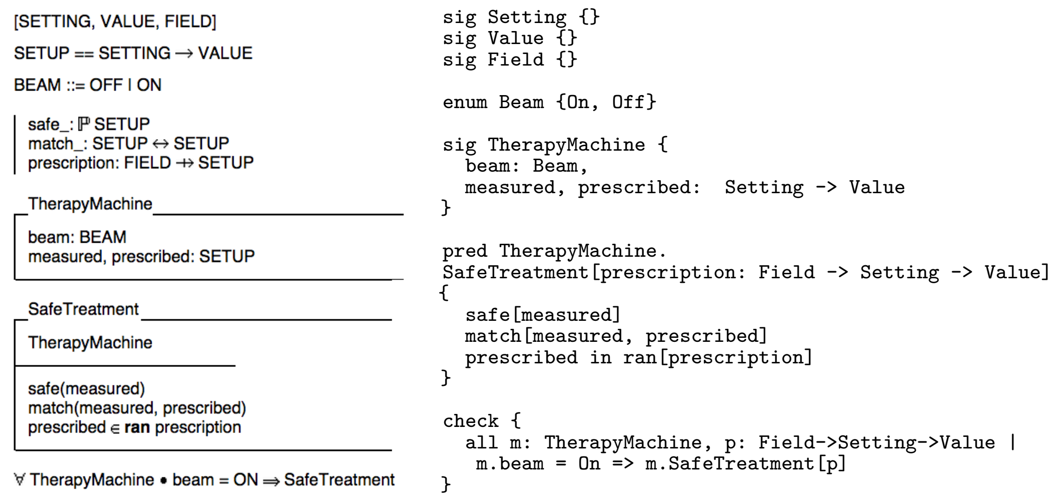
What form should a dependability case for CNTS—or any system—take? Jackson's proposal does not mandate any specifics, noting only that the level of detail and formality for a case will vary between systems. We argue that a dependability case should be *formal*—that is, specified in a formal *dependability case language* (DCL) and subject to formal reasoning. Such a language would bring the same benefits to dependability cases that specification languages bring to software design—precision of expression and thought, automation to guard against syntactic and (some) semantic errors, and support for maintenance as the system evolves and the CNTS staff changes.

What then should a DCL look like? Existing DCLs are either logic-based languages (e.g., [17, 8]) for formalizing claims or structured notations (e.g., [26]) for relating claims to evidence. The former support mechanical analysis but have no notion of evidence. The latter include the notions of clams and evidence but, as semi-formal notations, they are not amenable to mechanical analysis. We plan to develop a new hybrid DCL that provides both mechanical reasoning and a (semi-)mechanical means of connecting claims with various forms of evidence—such as tests, Coq proofs, solver-aided reasoning, and manual reasoning by domain experts (for assumptions about the environment that cannot be otherwise discharged).

The details of our DCL are still being developed, but some necessary requirements have already become clear. First, it must be *expressive* enough to capture both system-level requirements and code-level specifications, since a dependability case spans all layers of design. Second, it must be *analyzable*—the consistency of the overall argument should be checkable in an automated fashion. Third, it must include a notion of evidence and be *flexible* enough to admit heterogeneous evidence of dependability. Finally, it must be *accessible* to domain experts who should be able to audit (but not necessarily construct) a case expressed in the language. If we succeed in striking a balance among these features, we expect the final design to be akin to SRI's Evidential Tool Bus [7]—but with a richer semantics specialized to our target domain and to our workflow of tools for generating evidence.

3.3 A Dependability Case Workflow

While a DCL helps express dependability claims and check their consistency with evidence and each other, it does not, by itself, help with the production of evidence. For that, we propose an integrated workflow of complementary technologies: Coq for infrastructure



```
[SETTING, VALUE, FIELD]

SETUP == SETTING → VALUE

BEAM ::= OFF | ON

| safe_: ℙ SETUP
| match_: SETUP ↔ SETUP
| prescription: FIELD ⇸ SETUP

 _TherapyMachine_______________________
|
| beam: BEAM
| measured, prescribed: SETUP
|______________________________________

 _SafeTreatment______________________
|
| TherapyMachine
|_____________________
|
| safe(measured)
| match(measured, prescribed)
| prescribed ∈ ran prescription
|____________________________________

∀ TherapyMachine • beam = ON ⇒ SafeTreatment
```

(a)

```
sig Setting {}
sig Value {}
sig Field {}

enum Beam {On, Off}

sig TherapyMachine {
  beam: Beam,
  measured, prescribed:  Setting -> Value
}

pred TherapyMachine.
SafeTreatment[prescription: Field -> Setting -> Value]
{
  safe[measured]
  match[measured, prescribed]
  prescribed in ran[prescription]
}

check {
  all m: TherapyMachine, p: Field->Setting->Value |
   m.beam = On => m.SafeTreatment[p]
}
```

(b)

Figure 3 An example specification in Z (a) and Alloy (b). The therapy beam can only turn on or remain on when the actual setup of the machine matches a stored prescription that the operator has selected and approved.

verification; solver-aided tools for accelerating the development and inspection of application code; and Alloy for describing and checking the high-level requirements and design.

As a first step, we will formalize the EPICS$_0$ DSL [20] in Coq. Such a formalization is critical, since EPICS has no formal semantics. We will then make EPICS$_0$ *solver-aided* [34, 35]—that is, equipped with automatic verification and synthesis tools based on SAT and SMT solving. These tools are intended to accelerate the standard CNTS development workflow [20] by guiding code reviews (with lightweight verification) and revisions (with lightweight synthesis). Next, we will use Alloy to check the CNTS design for high-level dependability properties, leveraging the existing Z specification [19] for the system, which is yet to be subjected to fully automatic analysis. Finally, we will integrate all these sources of evidence—Coq proofs, automatic synthesis and verification guarantees, Alloy models, and tests—into a dependability argument in our DCL.

To illustrate the components of our workflow, consider the primary safety requirement of CNTS. Figure 3a shows a formal specification of this requirement in Z. The specification was written [19] and manually analyzed by the CNTS staff. In our workflow, the requirement would be expressed in Alloy, as shown in Figure 3b. Because Alloy was inspired by Z, the two formulations correspond closely to each other. An engineer who knows Z can switch to Alloy without much trouble and gain the benefits of its automated analysis.

In an end-to-end dependability case, the Alloy requirement would be decomposed further into assumptions about the environment and into specifications about the CNTS software controller. The former would be discharged manually by experts. The latter would be discharged (within finite bounds) by a solver-aided verifier for EPICS$_0$. Our verifier simply assumes that the EPICS$_0$ implementation is correct. Our dependability case, however, would treat this assumption as a correctness claim to be discharged with Coq.

Figure 4 shows an example theorem in Coq for the correctness of the EPICS$_0$ implementation. This theorem requires that the interpreter function `interp` behaves according to the EPICS semantics captured by the `step` relation. In particular, whenever `interp` executes an EPICS *database* `db` (which includes both application code and state) with the inputs

```
Theorem interp_ok: forall db db' es,
    interp db (inputs es) = (db', outputs es) -> star step db es db'.
```

Figure 4 An example correctness theorem in Coq.

of some event trace `es`, denoted by `(inputs es)`, and produces the resulting database `db'` with output events `(outputs es)`, then the EPICS small step operational semantics allow exactly the same state transitions with the same observable input and output events. This theorem completes the dependability argument since the solver-aideded verifier (which is part of our trusted code base) uses the semantics specified by the `step` relation.

Our workflow is chosen to balance the need for high assurance with the cost of obtaining it. At the infrastructure level, the high cost of verifying rich properties manually in Coq is appropriate, since an infrastructure bug can invalidate any guarantees established for applications and because, once verified, the infrastructure changes slowly relative to application code. At the application level, however, code is continuously evolving, especially in a long-lived system such as CNTS. At this level, full automation is invaluable—the main purpose of tools is to accelerate development rather than provide total guarantees [14]. Automation is also critical at the design level, where bugs have the most serious effects [15]. We have chosen Alloy for this purpose because of its Z-like relational logic, fully automated analysis, and history of successful applications to safety-critical designs (e.g., [10, 32, 25]).

4 Related Work

There has been much prior work on building dependable systems, most of it by the software engineering community. Thanks to this work, we have effective approaches to gathering and analyzing whole-system requirements (e.g., [18]); to specifying and analyzing software designs (e.g., [33, 28, 16]); and, to turning those designs into analysis-friendly low-defect code (e.g., [14, 13, 19, 20]). The recent work on dependability cases (e.g., [6, 15, 31]) also gives us a methodology for driving the system-building process in a goal-directed fashion, so that the output of the process is both the system itself and an end-to-end argument—a social proof—that the system satisfies its critical requirements.

We aim to build on this body of work, and to produce the first integrated workflow—from languages to tools—for the development, evolution, and maintenance of a high-value radiotherapy system. In particular, we plan to leverage and extend existing work on languages for describing dependability cases [17, 8, 26] and for integrating heterogeneous tools into a workflow [7]. Unlike these prior languages, our DCL will be specialized to a narrow target domain. As such, it will be more accessible to domain experts, and more tightly integrated with the tools in our workflow—Coq [5], Alloy [16], and solver-aided verification and synthesis [34, 35].

5 Conclusion

We have outlined a research agenda to bring modern ideas in programming-language verification to the next-generation development of a safety-critical medical tool. Our approach is unusual in aiming first at one specific system before succumbing to the obvious temptation of building general tools for a large class of similar systems. Indeed, the goal of building a safe system will need to take priority over the goal of finding novel programming-languages research questions, but we believe there will be no shortage of the latter. We are optimistic that we will learn lessons that can inform research in several areas, namely interactive theorem proving, solver-aided languages, finite-system modeling tools, and—tying them together—formal languages for expressing dependability cases for safety-critical systems.

References

1 John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley, April 2003.

2 Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL)*, pages 90–101, Savannah, GA, January 2009.

3 Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207, San Diego, CA, June 2003.

4 Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, San Jose, CA, June 2011.

5 Coq development team. *Coq Reference Manual, Version 8.4pl5.* INRIA, October 2014. `http://coq.inria.fr/distrib/current/refman/`.

6 National Research Council, Daniel Jackson, and Martyn Thomas. *Software for Dependable Systems: Sufficient Evidence?* National Academy Press, Washington, DC, USA, 2007.

7 Simon Cruanes, Grégoire Hamon, Sam Owre, and Natarajan Shankar. Tool integration with the evidential tool bus. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *VMCAI*, volume 7737 of *Lecture Notes in Computer Science*, pages 275–294. Springer, 2013.

8 Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. In *Selected Papers of the Sixth International Workshop on Software Specification and Design*, 6IWSSD, pages 3–50, Amsterdam, The Netherlands, The Netherlands, 1993. Elsevier Science Publishers B. V.

9 Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22(5):271–280, May 1979.

10 Greg Dennis, Robert Seater, Derek Rayside, and Daniel Jackson. Automating commutativity analysis at the design level. In *Proceedings of the 2004 International Symposium on Software Testing and Analysis*, pages 165–174, Boston, MA, July 2004.

11 EPICS. `http://www.aps.anl.gov/epics/`.

12 Galois. `http://galois.com`.

13 Gerard J. Holzmann. The power of 10: Rules for developing safety-critical code. *Computer*, 39(6):95–97, June 2006.

14 Gerard J. Holzmann. Mars code. *Commun. ACM*, 57(2):64–73, February 2014.

15 Daniel Jackson. A direct path to dependable software. *Commun. ACM*, 52(4):78–88, April 2009.

16 Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press, February 2012.

17 Daniel Jackson and Eunsuk Kang. Property-part diagrams: A dependence notation for software systems. Technical report, Massachusetts Institute of Technology, 2009. `http://hdl.handle.net/1721.1/61343`.

18 Michael Jackson. *Problem Frames: Analysing and Structuring Software Development Problems.* Addison-Wesley, 2001.

19 Jonathan Jacky. Formal safety analysis of the control program for a radiation therapy machine. In Wolfgang Schlegel and Thomas Bortfeld, editors, *The Use of Computers in Radiation Therapy*, pages 68–70. Springer Berlin Heidelberg, 2000.

20 Jonathan Jacky. EPICS-based control system for a radiation therapy machine. In *International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS)*, 2013.

21 Jonathan Jacky and Ruedi Risler. Clinical neutron therapy system reference manual. Technical Report 99-10-01, University of Washington, Department of Radiation Oncology, 2002.

22 Jonathan Jacky and Ruedi Risler. Clinical neutron therapy system therapist's guide. Technical Report 99-07-01, University of Washington, Department of Radiation Oncology, 2002.

23 Jonathan Jacky, Ruedi Risler, David Reid, Robert Emery, Jonathan Unger, and Michael Patrick. A control system for a radiation therapy machine. Technical Report 2001-05-01, University of Washington, Department of Radiation Oncology, 2001.

24 Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing browser security guarantees through formal shim verification. In *Proceedings of the 21st Usenix Security Symposium*, pages 113–128, Bellevue, WA, August 2012.

25 Eunsuk Kang and Daniel Jackson. Formal modeling and analysis of a flash filesystem in Alloy. In *Proceedings of the 1st International Conference on Abstract State Machines, B and Z*, ABZ '08, pages 294–308, Berlin, Heidelberg, 2008. Springer-Verlag.

26 Tim Kelly and Rob Weaver. The goal structuring notation – a safety argument notation. In *Proceedings of the Dependable Systems and Networks 2004 Workshop on Assurance Cases*, July 2004.

27 Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Michael Norrish, Rafal Kolanski, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, Big Sky, MT, October 2009.

28 Kevin Lano. *The B Language and Method: A Guide to Practical Formal Development.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1996.

29 Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 216–226, Edinburgh, UK, June 2014.

30 Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

31 Joseph P. Near, Aleksandar Milicevic, Eunsuk Kang, and Daniel Jackson. A lightweight code analysis and its role in evaluation of a dependability case. In *Proceedings of the 33rd International Conference of Computer Safety, Reliability and Security*, pages 31–40, Waikiki, Honolulu, HI, May 2011.

32 Tahina Ramananandro. Mondex, an electronic purse: Specification and refinement checks with the alloy model-finding method. *Form. Asp. Comput.*, 20(1):21–39, December 2007.

33 J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

34 Emina Torlak and Rastislav Bodik. Growing solver-aided languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 135–152, Indianapolis, IN, 2013.

35 Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 530–541, Edinburgh, UK, June 2014.

36 Richard Uhler and Nirav Dave. Smten with satisfiability-based search. In *Proceedings of the 2014 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 157–176, Portland, OR, October 2014.

37 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 283–294, San Jose, CA, June 2011.

38 Pamela Zave. Using lightweight modeling to understand Chord. *SIGCOMM Comput. Commun. Rev.*, 42(2):49–57, March 2012.