Investigating Safety of a Radiotherapy Machine Using System Models with Pluggable Checkers

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

[1]University of Washington, Department of Computer Science, Seattle, USA
{spernste, loncaric, emina, ztatlock, xi, mernst}@cs.washington.edu
[2]University of Washington, Department of Radiation Oncology, Seattle, USA
jon@uw.edu

Abstract. Formal techniques for guaranteeing software correctness have made tremendous progress in recent decades. However, applying these techniques to real-world safety-critical systems remains challenging in practice. Inspired by goals set out in prior work, we report on a large-scale case study that applies modern verification techniques to check safety properties of a radiotherapy system in current clinical use. Because of the diversity and complexity of the system's components (software, hardware, and physical), no single tool was suitable for both checking critical component properties and ensuring that their composition implies critical system properties. This paper describes how we used state-of-the-art approaches to develop specialized tools for verifying safety properties of individual components, as well as an extensible tool for composing those properties to check the safety of the system as a whole. We describe the key design decisions that diverged from previous approaches and that enabled us to practically apply our approach to provide machine-checked guarantees. Our case study uncovered subtle safety-critical flaws in a pre-release of the latest version of the radiotherapy system's control software.

Keywords: case study, safety-critical systems, SMT-based verification, lightweight formal methods

1 Introduction

Formal techniques for guaranteeing software correctness have made tremendous progress in recent decades. However, applying these techniques to real-world safety-critical systems remains challenging for three reasons. First, using general-purpose tools to formally prove deep properties of a system component (e.g., functional correctness of a cryptographic primitive [6]) requires substantial expertise and manual effort. Second, many real systems contain components for which effective formal analysis is still an active research topic (e.g., formally guaranteeing liveness for a consensus protocol within a distributed system [22]), and thus in practice, these components can only be analyzed by weaker techniques such as testing or

expert review. Third, even when deep properties can be established for individual system components, their composition may not add up to overall system safety, leading to catastrophic failures [25].

This paper reports on a large-scale case study in applying modern verification techniques to check the safety of a radiotherapy system in current clinical use: the Clinical Neutron Therapy System (CNTS) at the University of Washington Medical Center. We describe how to practically address the above challenges with a combination of techniques that reason at the system and component levels, and that provide guarantees of varying strength, from automatic proof to manual review. To check system-level properties (such as "the beam shuts off if physical settings of the machine do not match the prescription"), we developed and analyzed a formal model of CNTS in Alloy [1, 26]. Using this model, we obtained partial specifications of critical component-level properties (such as "the therapy control software sends a shut-down message if it receives an out-of-tolerance sensor reading"). To check the resulting component properties, we built a suite of custom tools, ranging from an SMT-based verifier for properties of the control software to a manual review processor for properties of the physical components. These custom tools plug into our *safety case checker* (SCC), which combines the system model with the results of the component checks into a mechanized argument—a form of a *safety case* [28, 48]—that CNTS satisfies its critical safety properties.

To the best of our knowledge, this case study presents the *first formal, machine-checked safety case for a real system.* A typical safety case takes the form of a structured argument, expressed in natural language or graphical notation [14, 37], that a system satisfies a desired critical property. This argument decomposes the system property into a set of component properties, each of which is justified with concrete *evidence* (e.g., the results of verification, manual review, or testing). As massive informal artifacts, however, typical safety arguments suffer from logical fallacies [21] and are difficult to audit for the presence and sufficiency of evidence. For these reasons, prior work [42, 43] has called for mechanization of safety case checking. Our work shows, for the first time, that such mechanization is not only possible, but that it is both practical and useful. Building on existing verification frameworks [1, 46], our safety case (including the system model and the component checkers) consists of just 2700 lines of code, yet its checks uncovered safety-critical flaws in pre-release versions of the CNTS control software.

Our work differs from prior efforts [10] at safety case mechanization in that *the safety argument is embedded into a formal model of the system.* In prior work, properties of system components are abstracted by uninterpreted predicates, and a mechanized tool ensures that the logical structure of the argument is sound. With this approach, missing properties or assumptions about the environment are discovered through manual auditing. We take a different approach, which enables us to both check the logical soundness of the argument and mechanically discover (some kinds of) missing properties. In particular, our safety argument takes the form $S \Rightarrow P$, where S is a model of the system and P is the desired safety property. The model S specifies the behavior of the system in terms of the properties $s_1, \ldots, s_n$ of individual components and their interactions. When

expressed in Alloy, such a model enables both bounded simulation and checking of abstract executions of the system. Bounded simulation helps ensure that the model S includes all desirable treatment scenarios (thus guarding against vacuous fulfillment of the safety argument $S \Rightarrow P$). The checking, on the other hand, helps detect missing component properties (e.g., "the Ethernet network does not drop messages") that are necessary to establish the safety property P.

A component property s_i serves as a (partial) specification for the implementation of the component c_i in the underlying system. Ideally, each component c_i would be mechanically verified to satisfy s_i, but such verification is currently infeasible for many properties (e.g., the reliability of an Ethernet network). Our safety case therefore employs a pragmatic approach that allows weaker evidence to be used for these properties. In particular, each property s_i is guarded by an uninterpreted predicate, written as $evidence(tool, args) \Rightarrow s_i$, which states that the given external tool establishes the property s_i for the component c_i. When checking a safety case, SCC invokes the specified external tools to determine the truth of these predicates, but it does not reason about the strength of the evidence provided by the tool: the *sufficiency* of evidence is subject to manual auditing. Our safety case aids this manual part of the process by making explicit, and precise, all links from a safety argument to evidence.

The rest of this paper is organized as follows. Section 2 provides an overview of the CNTS system. Section 3 describes our machine-checked safety case for CNTS. Section 4 presents SCC and other tools we built as part of the case study. Section 5 discusses the flaws that these tools helped us find and correct. Section 6 surveys related work, and Section 7 concludes the paper.

2 Overview of CNTS

The Clinical Neutron Therapy System (CNTS) is a radiotherapy machine that uses neutron radiation to treat tumors resistant to conventional radiotherapy. Due to the high installation and maintenance costs of neutron-based systems, CNTS has been in service for over 30 years, and it is one of only three systems of this kind in the United States. As such, it depends heavily on custom software developed by CNTS engineers, who have achieved a remarkable safety record, with no serious misadministrations due to machine or control system problems.

CNTS engineers have re-implemented its therapy control software twice since 1984 [30, 35]. The latest controller [31] is implemented in a subset of the EPICS (Experimental Physics and Industrial Control System) dataflow language [17], which is widely used for controlling scientific instruments. This development effort aims to support new therapies, integrate new software and hardware, and adapt to changes in the hospital's systems—with continued safety as the foremost concern.

CNTS as a whole is designed [34] to enforce a number of critical safety properties, including *prescription safety*, which is the focus of our work:

> **Prescription Safety Property (P_{rx}).** *During treatment, the beam will turn off if any physical machine setting moves outside the tolerances*

specified by the prescription and the operator has not issued the manual override command.[1]

This property is enforced by a *control subsystem*, consisting of hardware and software components, that monitors and drives the system's physical components. Our safety case for P_{rx} (Section 3) spans the control subsystem of CNTS, as well as the physical components involved in a treatment prescription.

Physical Components. The key physical components of CNTS include the cyclotron, the leaf collimator, the gantry, and the treatment couch. The cyclotron generates a broad beam of particles, which passes through the leaf collimator in the gantry head on its way to the patient. The collimator consists of forty steel leaves that control the shape of the beam. The gantry head also contains a set of wedges and filters that can be inserted to further adjust the beam's shape and intensity. The gantry rotates 360 degrees around the treatment couch so that the beam can enter the patient from any angle. The couch itself has five degrees of motion freedom. A treatment prescription specifies settings for all of these components, and the CNTS control subsystem ensures that each setting remains within prescribed tolerances during treatment.

Control Subsystem. The CNTS control subsystem is a collection of hardware and software components, which communicate by exchanging messages through a private Ethernet network. The components relevant to our safety case include:

Embedded Single-Board Computers interface with motors and sensors for controlling the physical components of the system. For example, the Treatment Motion Controller (TMC) monitors the orientation of the couch and the gantry. Separate computers exist for shaping the neutron beam and for monitoring the patient's radiation exposure. These embedded computers were installed by the original system vendor and are treated as black boxes by CNTS engineers.

Therapy Control (TC) Software displays the user interface on the operator console, accepts commands from the operator, coordinates the activity of the embedded computers, and helps enforce CNTS safety properties (such as P_{rx}). The TC, now re-implemented in EPICS, runs on a general-purpose Linux computer.

Programmable Logic Controller (PLC) serves as an interface between the networked components and the cyclotron control hardware. The PLC is electrically connected to the Hardwired Safety Interlock System (HSIS), which consists of a series of mechanical relays. These relays carry power to the radio frequency amplifiers that accelerate particles in the cyclotron. If any relay is opened, the cyclotron stops receiving power and the neutron beam shuts off. For example, if a machine setting moves out of the prescribed tolerances, the TC sends a signal to the PLC to open an HSIS relay, thus shutting off the beam. The PLC functionality is implemented using ladder logic [36], and the CNTS staff modifies and maintains this implementation in addition to the TC.

[1] The manual override exists to enable treatment to continue if a sensor for a machine setting generates a false alarm.

3 Building a Case for Prescription Safety

This section describes our approach to building a formal, machine-checkable safety case for a key property of CNTS—prescription safety (P_{rx}). The presentation focuses on the aspects of case design that enabled us to practically and effectively check a deep property of a complex system. The tools we developed, and the results we obtained, are discussed in the following sections. All artifacts comprising the case can be found on the project's web page [3].

3.1 Structuring the Case: A System Model with Pluggable Checkers

Our safety case for P_{rx} consists of two parts: (1) a system model S that specifies partial properties of the components spanned by P_{rx} (Section 2), and (2) a set of custom tools for checking if those components satisfy the specified properties. The tools indicate either success or failure, depending on whether a given property can be established (through some means) for a given component. Our safety case checker (SCC) connects the model and the tools through uninterpreted predicates of the form $evidence(tool, args)$, which guard component properties specified by S (see, e.g., Figure 2). These predicates tell SCC which tools to invoke, and how, to establish a given property. The SCC collects pass/fail results of running the tools and uses an off-the-shelf counterexample finder [1] to check that, assuming the obtained results, $S \Rightarrow P_{rx}$. A safety case constructed in this way provides a high-level safety argument (i.e., the system model S, the system property P_{rx}, and the predicates describing how the evidence is obtained), which aids manual auditing, as well as a formal artifact, which enables automatic checking for (some classes of) safety-critical regressions as the system evolves.

3.2 Making the Case: Diagrams, Models, and Tools

We developed the safety case for P_{rx} in three steps. First, we worked with CNTS engineers to identify the system components and interactions that are relevant to P_{rx}, producing an informal, diagrammatic model of the system. This step was crucial for determining the level of detail at which to model the system, and to understand the engineers' concerns—specifically, which properties mattered most and where to focus our analysis effort. Next, we formalized and refined the system model in Alloy [1, 26], a widely-used specification language that extends first-order logic with transitive closure and relational algebra. Throughout the formalization process, we relied heavily on Alloy's counterexample finder to detect errors in logical reasoning, as well as component properties that were missing from our informal diagrams. Finally, we used the resulting Alloy model—and feedback from CNTS engineers—to determine where and how to focus our tool-building effort. We describe each of these steps in more detail below.

Drafting an informal case. To build a safety case for P_{rx}, we first drafted an informal safety argument, expressed as a property-part diagram [27]. This kind of diagram shows how components—or parts—of the system relate to each

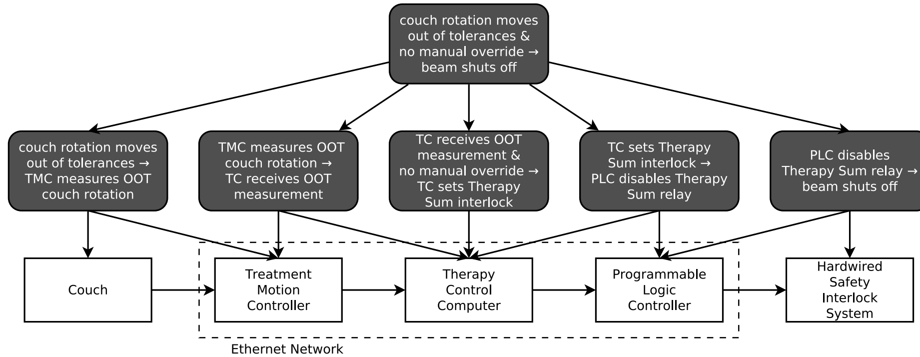

Fig. 1. A fragment of the property-part diagram for a part of the P_{rx} property. Boxes represent system components, and rounded boxes represent properties. Edges originating from a property indicate that the property depends on the target components or properties for its fulfillment. Edges originating from a component indicate interaction via messages. Components within the dashed box are connected via Ethernet.

other and how their individual properties collectively satisfy the desired system property. Figure 1 shows a fragment of the property-part diagram for P_{rx} that covers the rotation angle of the treatment couch. The diagram shows how the top-level property and its sub-properties are established by a combination of other properties and components. For example, the Programmable Logic Controller (PLC) and the Hardwired Safety Interlock System (HSIS) jointly ensure that the beam is turned off when the Therapy Sum relay is opened (disabled). The informal case for P_{rx} includes similar diagrams for other machine parameters that are specified by a treatment prescription.

Formalizing the case. To formalize the P_{rx} case, we developed an Alloy model of the major components of CNTS. These include the therapy control (TC) software, the Treatment Motion Controller (TMC), the Programmable Logic Controller (PLC), the Hardwired Safety Interlock System (HSIS), the beam, and the machine settings involved in a prescription. Our model specifies the internal states of the components (e.g., whether a given HSIS relay is open or closed) and their external interactions (e.g., the messages exchanged between the TC and the embedded computers) in sufficient detail to capture the key behaviors of the system, such as the beam shutting off when a machine setting enters an out-of-tolerance state, or the beam staying on due to manual override. The lead CNTS engineer audited the model to ensure that it accurately describes the partial properties of the system relevant to the argument.

Figure 2 shows a snippet of our formalization. Lines 7–13 specify a property of the PLC that is relevant to the case. We model the changes in the state of the PLC relays and coils as events ordered by the `next` relation. Relays can be either open or closed, and coils can be either energized or de-energized. Relay 2754 of the PLC corresponds to the Therapy Sum Interlock relay in Figure 1. Whenever this relay is in the open state (line 7), PLC coil 1623 enters a de-energized state, which,

in turn, electrically signals an HSIS relay to open and cause the beam to shut off (lines 8–13). Other parts of the system are modeled at a similar level of detail.

```
1  evidence[PLC_Analysis,
2           "--mode" -> "all-paths-to-coil-contain-relay" +
3           "--network-file" -> "plc-code/cyclotron/mod1.stu" +
4           "--coil" -> "%M1623" +
5           "--relay" -> "%M2754",
6           Proof] =>
7  all relayOpen: Relay2754.state & RelayOpen |
8    some coilState: Coil1623.state & CoilDeenergized,
9         coilChangeSignal : PLC.sentMsgs & CoilChange |
10     coilState in relayOpen.next and
11     coilChangeSignal in coilState.next and
12     coilChangeSignal.coil = Coil1623 and
13     coilChangeSignal.state = coilState
```

Fig. 2. A property of the PLC, guarded by an evidence predicate. The property states that PLC coil 1623 is de-energized when relay 2754 is open. Coil 1623 controls the neutron beam and relay 2754 is written to by the Therapy Controller (TC)—this is the mechanism by which the TC shuts off the beam when a machine setting is out of its prescribed tolerance. The property (lines 7–13) is a formula in the Alloy language [1, 26]. The evidence predicate (lines 1–6) states that the external `PLC_Analysis` tool (Section 4) must be called to establish that the PLC satisfies this property.

```
1  check PrescriptionSafetyCase {
2    all ms: MachineState |
3      (properties and
4      some badSetting[ms] and
5      not badSettingOverriden[ms]) =>
6      some off: Beam.state & BeamOff |
7        happensBefore[ms, off]
8  }
9  for 3 but 10 Event, 2 int
```

Fig. 3. The Alloy formulation of the P_{rx} property. For all machine states, if all component properties (e.g., Figure 2) hold, a machine setting is out of prescribed tolerances in that state, and the manual override has not been enabled for that machine setting, then a “beam off” event must occur after the given state.

While formalizing the case, we relied heavily on the Alloy Analyzer [1] to discover soundness and vacuity errors in our argument. Checking that the model is sound (permits no behavior that violates P_{rx}) let us detect missing component properties (e.g., the Ethernet network does not drop messages), and checking that the model is not vacuous (permits some behaviors that satisfy P_{rx}) let us detect accidental contradictions in the formalization. To check for soundness

errors, we asked the Analyzer to verify that our model of CNTS implies P_{rx} for all event sequences of length up to ten, as shown in Figure 3. The bound of ten encompasses all known treatment scenarios and is thus large enough to prevent P_{rx} from being vacuously fulfilled. To check this, we asked the Analyzer to verify that a bound of ten events is sufficient to simulate treatment scenarios in which (1) the beam remains on because the machine settings remain within tolerance; (2) a setting goes out of tolerance, but the beam shut off is manually overridden; and (3) the beam shuts off due to an out-of-tolerance setting and the absence of manual override. All of these checks explore massive potential state spaces (on the order of 2^{2800} states), and all pass in seconds.

Building Tools. Having formalized the case, we used it to decide what tools to build in order to establish the specified component properties. In particular, we determined the tool interfaces by going through the model and systematically annotating all component properties with *evidence* predicates, as shown in Figure 2. An evidence predicate evaluates to true if and only if the specified tool indicates success when invoked with the given arguments. For example, lines 1–6 in Figure 2 state that the evidence (a proof) for the PLC property is produced by the PLC analysis tool (Section 4), invoked with the specified parameters. We determined what kind of evidence was practically sufficient for each component (e.g., a proof, passing tests, or manual review) based on the feedback from CNTS engineers. This process of connecting properties with evidence was crucial in focusing our analysis effort (Section 5): building a tool that proves a narrow class of properties is much easier than building a general-purpose verifier. Our safety case both articulated these properties and helped guide the construction of tools for checking them.

4 Tools and Analyses

This section surveys the tools that we built to produce and check evidence for the P_{rx} safety case described in Section 3. We focused the analysis effort on the components that are directly modified by the CNTS engineers: the Therapy Control (TC) software and the Programmable Logic Controller (PLC). In particular, we built a linter and an SMT-based verifier for the subset of EPICS in which TC is written, a static analyzer for (a narrow class of) properties of PLC ladder logic code, and a static analyzer for the EPICS-PLC interface code. We relied on expert approval and manual review of documentation to establish the relevant properties of the embedded computers, the Hardwired Safety Interlock System (HSIS), the Ethernet network, and the physical components. The presence of this expert evidence was checked using a simple text-processing tool. We also built a safety case checker (SCC) to connect our formal model of CNTS (Section 3) with the results produced by the tools. We describe each of these tools in more detail below, highlighting how the safety case guided our choice of analyses, their construction, and their application.

Safety Case Checker (SCC). Our safety case checker (SCC) automates the integration of external evidence-generating tools into a formal safety argument, expressed with respect to a model of the system. We chose the Alloy language and the Alloy Analyzer as our toolset for system-level reasoning. However, a similar checker could also be created for other formal frameworks (e.g., [9, 40, 41]).

```
1  procedure SCCCheck(S, P, universe_size):
2    I[evidence] ← {} // partial interpretation
3    for evidence[tool, args, kind] in FindEvidencePredicates(S):
4      result ← InvokePluggableChecker(tool, args)
5      if result = true:
6          I[evidence] ← I[evidence] ∪ {⟨ tool, args, kind ⟩}
7      return AlloyCheckSAT(S ∧ ¬ P, I, universe_size)
```

Fig. 4. Overview of the algorithm used by our safety case checker.

The SCC consists of two parts: (1) an extension to the Alloy language, and (2) a checker for this language extension that is based on the Alloy Analyzer. The language is extended with a small library that provides the uninterpreted `evidence` predicate. In particular, the formula `evidence[tool, args, kind]` represents the presence (or absence) of the given kind of evidence, produced by invoking the specified tool on the provided arguments. The checker, as shown in Figure 4, provides an interpretation for each `evidence` predicate in a safety case by performing the specified tool invocation and recording the result in Alloy. This invocation is performed through a simple plug-in interface, designed for easy addition of new tools to the checker. The resulting interpretation I gives meaning only to the `evidence` predicates, and, as such, it is a *partial interpretation* [44, 47] for the safety case $S \Rightarrow P$, which includes additional relations (e.g., the `next` relation in Figure 2). The checker passes I and $S \wedge \neg P$ to the Alloy Analyzer, which checks that I cannot be extended into a finite counterexample to the safety case—i.e., an interpretation $I' \supseteq I$ that satisfies $S \wedge \neg P$ in a finite universe of discourse. The results of SCC verification are sound up to the bound on the universe size, modulo any bugs in the implementation of SCC and the pluggable checkers.

EPICS Verifier We focused our most advanced analysis, a fully automated verifier for a subset of the EPICS language, on the newest component of the system, the Therapy Control (TC) software. This tool uses an SMT solver to verify safety properties of programs written in a subset of EPICS that includes all code from CNTS. Because the Therapy Control software is finite-state, with bounded execution length and memory consumption, our EPICS verifier is both sound (it does not miss defects) and complete (it does not report false positives) for CNTS. The P_{rx} case uses the verifier to prove that the therapy control software initiates beam shut-off whenever it receives an out-of-tolerance reading from a sensor (see, e.g., Figure 5).

The verifier builds on Rosette [45], a language for constructing verification and synthesis tools based on SMT solvers. It takes as input an EPICS dataflow

program and a safety property, symbolically interprets the program on an arbitrary state and input event, and then invokes the Z3 solver [11] to check that the property holds in the resulting state. The output is either a guarantee that the property holds, or a concrete state and input event (a counterexample) that causes the program to violate the property.

An EPICS dataflow program is a graph consisting of edges, called *links*, and nodes, called *records*, along with configuration settings and initial values for each record. At run time, the EPICS interpreter behaves as a reactive system, responding to *events* (such as arrival of input from external devices or expiration of a timer) by updating the state of the graph and possibly sending output to external devices. In contrast to traditional dataflow systems, which automatically update all dependent values when an input value changes, the EPICS language gives the programmer explicit control over all data flow and control flow in the program. Moreover, EPICS code can modify the structure of the dataflow graph at run time, adding or removing edges, as well as modifying record configurations, such as the expression evaluated in a record. These dynamic features make it challenging to verify general EPICS code.

Unlike general EPICS programs, however, CNTS code is amenable to verification because it does not use the dynamic features of EPICS. Since EPICS forbids dynamic memory allocation and guarantees termination of event handlers, all EPICS programs with no dynamic features are finite-state, and their executions have bounded length. As a result, our verifier can automatically prove deep properties of CNTS code, without requiring loop invariants or imposing artificial bounds on heap size and execution length. Our verifier uncovered a safety-critical bug in a production version of the CNTS therapy control software, in addition to a subtle dependence of a pre-release version of the software on a bug in the EPICS runtime, as described in Section 5.1.

```
1 (define (couch-wrong-implies-beam-off)
2   (process_IsoGantryCouchTurntableActual)
3   (assert (=>
4     (and
5       (> (abs (- prescribed actual)) tolerance)
6       (= couch-override 0)
7       (= mode 0))
8     (= beam-interlock 0))))
```

Fig. 5. A property of the CNTS therapy control software verified with the EPICS verifier. It states that, after a couch turntable angle reading is processed (line 2), the beam interlock is triggered (“beam off”) if the couch turntable’s actual rotation differs from the prescription by more than the tolerance, the manual override is disabled, and the machine is in therapy mode.

EPICS Linter The EPICS linter establishes a basic well-formedness property assumed by our verifier: all record links (i.e., dataflow edges) in a program refer to valid records. EPICS does not otherwise report broken links, instead assuming

that missing records will be provided by other EPICS instances on the same local network. Ensuring that all links are valid implies that a given EPICS program contains all code relevant to the analysis. Our linter uncovered several issues in the therapy control code, described in Section 5.1.

EPICS-PLC Interface Checker The EPICS-PLC interface checker ensures that a given EPICS record is connected to a particular PLC relay. This requires analyzing both the EPICS program and a separate startup script that initializes communication with the PLC. The P_{rx} safety case, for example, uses the interface checker to ensure that the EPICS Therapy Sum Interlock record is properly connected to the PLC Therapy Sum Interlock relay.

PLC Checker The PLC checker analyzes the graph of connections between coils, relays, and the power source in a ladder logic program. This checker provides two analyses that are useful for the P_{rx} case. First, it can check that a named relay's state is not updated by any other element within the PLC. Second, it can check that all paths from the power source to a named coil pass through a named relay, thus guaranteeing that the coil is energized only when the relay is closed. In the P_{rx} safety case, these checks establish that the Therapy Sum Interlock relay in the PLC is modified only as a result of messages from the therapy control software, and that opening the Therapy Sum Interlock relay must de-energize the PLC coil connected to the Hardwired Safety Interlock System (HSIS).

Expert Evidence The expert evidence tool allows an expert to assert that a component property holds based on manual inspection of some part of the system. After examining the property in question, the expert creates a text document explaining in prose the nature of the inspection and the evidence that supports the property, along with the expert's name and the current date to support future auditing. For example, the configuration of the CNTS HSIS is defined by a non-machine-readable circuit diagram, so the P_{rx} safety case relies on expert inspection to establish properties of the HSIS. In general, our case relies on expert evidence only for claims that would be impractical to support otherwise.

5 Results and Discussion

This section presents the results of developing a mechanically-checkable safety case for P_{rx}, and the lessons learned from our experience. Developing the case uncovered several issues in the new therapy control software for CNTS, including two safety-critical defects. We also found that structuring the case as a system model with pluggable checkers focused our analysis effort, enabling us to perform deep checks of a complex system, while writing only 2700 lines of code.

5.1 Issues Uncovered

Through the construction of the P_{rx} case, we found several previously unknown issues in a pre-release version of the CNTS therapy control software, and we rediscovered an issue that the CNTS engineers found during a production test

run. We discuss these issues first, and conclude by briefly describing a problem that we found in the system model itself.

Array Semantics While developing the part of the case related to the setting of the couch rotation angle, we discovered a serious issue in the therapy control code and in a major component of the EPICS runtime. The issue concerns array calculations, which are performed using EPICS records of type `acalcout`. An `acalcout` record performs calculations over arrays of a statically specified length. All intermediate values in the computation are truncated or padded with zeros to match this length.

The affected calculation in the therapy control software uses an "in-place slice" operator that retains elements between two given indices and zeroes the rest. In the documentation [16], the bounds on this operator are both inclusive, but in the version of `acalcout` used in CNTS, the upper bound is erroneously treated as exclusive. The therapy control software behaves correctly under the exclusive semantics, but upgrading to a version of `acalcout` that correctly implements the inclusive semantics would introduce subtle errors into several calculations in the therapy control software. Our EPICS verifier, initially implemented using the inclusive semantics, detected one such error in the computation of a flag sent to the PLC to control the neutron beam, for which *sending the wrong value may cause the beam to fail to turn off when a sensor reading is out of prescribed tolerances.* Due to this issue, the CNTS engineers cannot safely upgrade this essential library—correct behavior of the software depends upon the library bug. The CNTS engineers were unaware of this problem until our EPICS verifier revealed it.

Gantry Rotation Early in the first production run of the new therapy control software, the CNTS engineers identified a safety-critical flaw in the checking of the gantry rotation angle. The gantry angle measurement ranges from $-0.5°$ to $+360.5°$, and the intent when developing the gantry rotation checks was to treat pairs of angles separated by 360° as equivalent, so that a rotation measurement of 360° would satisfy a prescription for 0°. However, an error in the arithmetic used in the check caused the system to treat as equivalent any pair of angles equally distant from 180°; for example, a measurement of 200° would satisfy a prescription for 160°. *This error could have allowed the beam to turn on and remain on with the gantry rotation set to an incorrect angle.*

Before we had completed the relevant portion of the P_{rx} case, the CNTS engineers notified us that the code contained a bug involving the gantry rotation angle, but provided no other details. When we completed the case, SCC detected an error: the EPICS verifier, when asked to prove that the therapy control software triggered the Therapy Sum Interlock upon receiving an out-of-tolerance gantry rotation measurement, instead produced a counterexample containing a rotation measurement and prescription value that erroneously failed to trigger the interlock. We investigated the relevant EPICS code to identify the root cause of the bug and confirmed the details with the EPICS engineers. After applying their fix to the CNTS EPICS code, SCC processed the P_{rx} safety case without errors.

Broken Links Our EPICS linter uncovered a total of 59 references to nonexistent records in the therapy control software. Three of these represented serious

problems: they were caused by a misspelled record name, which would *prevent the operator from being informed of certain error conditions.* The CNTS engineers fixed these issues promptly. Another three turned out to be harmless remnants from an earlier code removal. The remaining 53 links refer to records that should have been annotated as nonlocal, since they exist in a separate EPICS installation that is accessed transparently over the network at runtime. The CNTS engineers are investigating how to annotate these nonlocal links.

Case Error Our initial formalization of the P_{rx} case contained an error, arising from a misunderstanding about the design of the system: the case claimed that an invalid couch rotation would cause the therapy control software to open the PLC's Gantry/Couch Subsystem Interlock relay, and the PLC would then internally open its Therapy Sum Interlock relay. But while trying to formulate the corresponding `evidence` invocation, we were unable to find the PLC relay number for the Gantry/Couch Subsystem Interlock. In fact, no such relay exists. The therapy software, not the PLC, combines Gantry/Couch Subsystem and other interlocks to compute the Therapy Sum, and it sends the combined Therapy Sum Interlock state to the PLC. Our strategy of connecting every component property to concrete evidence directly led to our discovery of this modeling error.

5.2 Performance

Analysis	Time (s)
EPICS verifier	3183.9
EPICS linter	0.9
EPICS-PLC interface checker	0.4
PLC ladder logic checker	0.4
Alloy Analyzer	4.1
Total (including overhead)	3190.0

Codebase	Size
PLC	5222 nodes, 10870 edges
TC	5448 lines
P_{rx} case	645 lines
SCC library	85 lines
EPICS tools	1666 lines
PLC checker	298 lines

Fig. 6. Total analysis time for the P_{rx} case (left), sizes of the CNTS software components analyzed as part of the case (right, above the line), and sizes of the system model and tools that make up the case (right, below the line). The performance data was collected on a Debian 8 laptop with an Intel Core i7-4900MQ CPU running at 2.80GHz with 16GB RAM.

As shown in Figure 6, end-to-end checking of our safety case takes less than an hour. The figure also shows the sizes of the codebases processed by our analysis tools. The PLC software is developed in a non-textual representation, so we report its size in terms of nodes (relays, coils, and other logic elements) and edges (wires) in the ladder logic graph. The size of the therapy control codebase includes both the EPICS dataflow graph definitions and the startup script used to load the graph and initialize communication between the therapy software and external hardware. The system model for the P_{rx} case is specified in Alloy,

extended with the SCC `evidence` library. The EPICS tools (the linter, verifier, and connection checker) share a common codebase, so they are reported together. Most of the common codebase is related to parsing; only 57 lines are specific to the linter, 441 are specific to the EPICS verifier, and 61 are specific to the EPICS-PLC interface checker. The case, in total, consists of 2700 lines of code.

5.3 Lessons Learned

Developing our safety case for CNTS led to three primary insights.

1. System Model as Safety Case Our decision to base the case on a detailed system model rather than a propositional formula (with component properties represented by uninterpreted predicates) enabled us to detect errors in the case with minimal auditing effort. A missing property in an Alloy model results in a concrete counterexample (an execution of the system that violates the safety property), whereas a missing premise in a propositional argument can be detected only through manual auditing. For example, our safety case for P_{rx} initially failed to specify that the ethernet network does not drop any packets. The Alloy Analyzer produced a counterexample, showing a scenario in which a dropped packet led to a violation of P_{rx}. Using a detailed Alloy model helped us not only ensure that relevant component properties are stated, but also that our argument is not vacuous (because the system model has no executions).

2. Simple Safety Case Checking The decision to implement our safety case checker (SCC) on top of Alloy significantly eased the development burden while providing us with ready-made automated analysis and visualization facilities. Initially, we had planned to implement SCC using a custom language to handle reasoning about the model and external evidence, as proposed in previous work [10, 12, 18, 19]. However, as part of the development of SCC, we chose to first prototype it directly in Alloy. From this prototype, the general design pattern for evidence predicates emerged, allowing us to easily connect additional external checkers to SCC, leading to a suite of lightweight but effective tools for the P_{rx} case.

3. Deep and Narrow Custom Tools We found case-guided tool development to be highly effective since it focused our efforts on the important properties of each component. With only 1964 lines of code, we built four custom tools for CNTS that check a narrow class of properties each, as specified by our system model. The integration of these tools into SCC was eased by its simple plug-in architecture. To add a new tool, the tool developer simply registers a plugin, consisting of a small Python script that can invoke the tool and interpret its output to determine success or failure. In particular, the need for the EPICS-PLC connectivity checker became apparent only late in the development of the P_{rx} case, but because we had already implemented the plug-in architecture for SCC, we were able to both develop and integrate this checker in less than a day.

6 Related Work

There is a large body of literature on ensuring safety of critical systems (see, e.g., [28, 38] for a survey). This section surveys the most closely related work, focusing on previous safety efforts at CNTS, methodologies for ensuring safety of complex systems, languages for expressing safety cases, and symbolic techniques for checking properties of software components.

CNTS Safety The CNTS engineering staff has, over the years, produced a large collection of heterogeneous evidence [29] in support of CNTS safety properties, including P_{rx}. This includes a 200-page document detailing system requirements, developed in consultation with physicists and clinicians; a 2,100 line Z specification [30] of the Therapy Control software; a 16,000 LOC reference implementation of the Z specification in C (in use until July 2015); a 240-page reference manual [33]; a 43-page therapist guide [32]; and an extensive set of end-to-end testing protocols that are executed on a daily, weekly, monthly, and yearly basis. However, none of these prior efforts produced an explicit, mechanically checked safety argument. Our P_{rx} safety case is the first such argument to be created for CNTS, and its creation has already led to improvements in the new CNTS software.

Approaches to Safety Traditional approaches to system safety are *process-based*. Systems like CNTS or the Mars rover [24] are developed 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., [23]), manual code reviews, use of static analysis, and extensive testing. Process-based approaches are highly effective at producing low-defect code. However, they provide no explicit argument that the system as a whole satisfies critical properties.

Our work builds on *case-based* approaches to safety (e.g., [25, 28, 37, 39]). These approaches aim to produce an explicit argument [25] that links claims about component behavior to concrete evidence in the form of tests, proofs, manual reviews, etc. Our approach to developing the P_{rx} case is most closely related to that of Near et al. [39]. We also used property-part diagrams [27] to first develop an informal case, which we then formalized in Alloy [1, 26]. But Near et al. do not connect their system-level argument to evidence; instead they use an interactive analysis to produce evidence obligations only for the software component of their target system. Our approach, in contrast, uses SCC to connect the system model to evidence generated by a variety of tools, including a fully automatic code verifier. To our knowledge, our case study is also the first to analyze low-level Programmable Logic Controllers as well as the system's software.

Languages for Expressing Safety Cases Existing languages and tools for developing dependability cases (e.g., [13, 37]) focus on managing the structure of a case. In these languages, a safety case takes the form of a semi-formal argument expressed in a graphical notation [37], with safety claims linked to evidence from heterogeneous sources. SCC, in contrast, focuses on expressing the

safety argument with respect to a detailed formal model of the system, linked to tool-generated evidence. This approach enables automated reasoning about the logical correctness and non-vacuity of the safety argument.

In terms of automation, SCC is most closely related to the Evidential Tool Bus (ETB) [10]. The ETB is a general-purpose framework for tool integration, for scripting distributed workflows, and for connecting claims with supporting evidence. Its input language is a variant of Datalog. SCC shares with the ETB the idea of a semantics-neutral connection between claims and evidence, using uninterpreted predicates. In contrast to the ETB, however, SCC provides a more expressive formal language (with quantifiers and transitive closure), suitable for system modeling. SCC also provides, via Alloy, automatic soundness and vacuity checking, as well as facilities for counterexample visualization. We made heavy use of these features while developing the P_{rx} safety case.

Software Verification There is a wide variety of verification tools (e.g., [2, 4, 5, 7, 8, 15, 20, 49]) for general-purpose programming languages. These tools are hard to build, requiring significant effort and expertise. As a result, they are rarely created for more specialized languages, such as EPICS. Our EPICS verifier is, to our knowledge, the first of its kind. Its implementation leverages Rosette [45, 46], a language designed for easy creation of domain-specific verification and synthesis tools based on SMT. Our verifier scales to real EPICS programs and is capable of finding subtle flaws that cannot be found without symbolic reasoning.

7 Conclusion

This paper reported on a case study in applying modern verification techniques to construct the first mechanically-checked safety case for a real safety-critical system, the Clinical Neutron Therapy System (CNTS). Our safety case includes a detailed formal model of CNTS and a set of tools for establishing component properties specified by the model. Leveraging existing formal tools, Alloy and Rosette, we built the entire case by writing just 2700 lines of code. The construction of the case revealed serious flaws in the CNTS therapy control software and in the implementation of the EPICS language, which we reported to the CNTS staff. Our results demonstrate that formal, checkable safety cases can provide significant practical benefits by focusing analysis effort on deep properties of system components that matter for the safety of the system as a whole.

Acknowledgments

This material is based on research sponsored by DARPA under agreement numbers FA8750-12-2-0107, FA8750-15-C-0010, and FA8750-16-2-0032. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

Bibliography

[1] Alloy: a language and tool for relational models. http://alloy.mit.edu/alloy/, 2014.

[2] Java PathFinder. http://babelfish.arc.nasa.gov/trac/jpf, 2015.

[3] Safety case for CNTS prescription safety. http://neutrons.uwplse.org, Aug. 2015.

[4] D. Babic and A. J. Hu. Calysto: scalable and precise extended static checking. In *ICSE*, 2008.

[5] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS*, 2004.

[6] L. Beringer, A. Petcher, K. Q. Ye, and A. W. Appel. Verified correctness and security of openssl hmac. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 207–221, Washington, D.C., 2015. USENIX Association. ISBN 978-1-931971-232. URL https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/beringer.

[7] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, San Diego, CA, Dec. 2008.

[8] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, 2004.

[9] Coq development team. *Coq Reference Manual, Version 8.4pl5*. INRIA, Oct. 2014. http://coq.inria.fr/distrib/current/refman/.

[10] S. Cruanes, G. Hamon, S. Owre, and N. Shankar. Tool integration with the evidential tool bus. In R. Giacobazzi, J. Berdine, and I. Mastroeni, editors, *VMCAI*, volume 7737 of *Lecture Notes in Computer Science*, pages 275–294. Springer, 2013.

[11] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78799-2, 978-3-540-78799-0. URL http://dl.acm.org/citation.cfm?id=1792734.1792766.

[12] E. Denney and G. Pai. Evidence arguments for using formal methods in software certification. In *Software Reliability Engineering Workshops (ISSREW), 2013 IEEE International Symposium on*, pages 375–380, Nov 2013. doi: 10.1109/ISSREW.2013.6688924.

[13] E. Denney, G. Pai, and J. Pohl. AdvoCATE: An assurance case automation toolset. In *Proceedings of the 2012 International Conference on Computer Safety, Reliability, and Security*, SAFECOMP'12, pages 8–21, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-33674-4. doi: 10.1007/978-3-642-33675-1_2. URL http://dx.doi.org/10.1007/978-3-642-33675-1_2.

[14] Dependability Research Group. Safety cases repository. http://dependability.cs.virginia.edu/info/Safety_Cases:Repository, February 2006.

[15] J. Dolby, M. Vaziri, and F. Tip. Finding bugs efficiently with a SAT solver. In *FSE*, 2007.

[16] EPICS. CalcOut Release Notes. http://www.aps.anl.gov/bcda/synApps/calc/calcReleaseNotes.html, .

[17] EPICS. EPICS, . http://www.aps.anl.gov/epics/.

[18] M. D. Ernst, D. Grossman, J. Jacky, C. Loncaric, S. Pernsteiner, Z. Tatlock, E. Torlak, and X. Wang. Toward a dependability case language and workflow for a radiation therapy system. In *Summit on Advances in Programming Languages (SNAPL)*, 2015.

[19] A. Gacek, J. Backes, D. D. Cofer, K. Slind, and M. Whalen. Resolute: An assurance case language for architecture models. *CoRR*, abs/1409.4629, 2014. URL http://arxiv.org/abs/1409.4629.

[20] J. P. Galeotti. *Software Verification Using Alloy.* PhD thesis, University of Buenos Aires, 2010.

[21] W. S. Greenwell, J. C. Knight, C. M. Holloway, and J. J. Pease. A taxonomy of fallacies in system safety arguments. In *International System Safety Conference*, 2006.

[22] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 1–17, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3834-9. doi: 10.1145/2815400.2815428. URL http://doi.acm.org/10.1145/2815400.2815428.

[23] G. J. Holzmann. The power of 10: Rules for developing safety-critical code. *Computer*, 39(6):95–97, June 2006. ISSN 0018-9162. doi: 10.1109/MC.2006.212. URL http://dx.doi.org/10.1109/MC.2006.212.

[24] G. J. Holzmann. Mars code. *Commun. ACM*, 57(2):64–73, Feb. 2014. ISSN 0001-0782. doi: 10.1145/2560217.2560218. URL http://doi.acm.org/10.1145/2560217.2560218.

[25] D. Jackson. A direct path to dependable software. *Commun. ACM*, 52(4):78–88, Apr. 2009.

[26] D. Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press, Feb. 2012.

[27] D. Jackson and E. Kang. Property-part diagrams: A dependence notation for software systems. Technical report, Massachusetts Institute of Technology, 2009. http://hdl.handle.net/1721.1/61343.

[28] D. Jackson and M. Thomas. *Software for Dependable Systems: Sufficient Evidence?* National Academy Press, Washington, DC, USA, 2007. ISBN 0309103940, 9780309103947.

[29] J. Jacky. The Clinical Neutron Therapy System. http://staff.washington.edu/jon/cnts/index.html.

[30] J. Jacky. Formal safety analysis of the control program for a radiation therapy machine. In W. Schlegel and T. Bortfeld, editors, *The Use of*

Computers in Radiation Therapy, pages 68–70. Springer Berlin Heidelberg, 2000.

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

[32] J. Jacky and R. Risler. Clinical neutron therapy system therapist's guide. Technical Report 99-07-01, University of Washington, Department of Radiation Oncology, 2002.

[33] J. Jacky and R. Risler. Clinical neutron therapy system reference manual. Technical Report 99-10-01, University of Washington, Department of Radiation Oncology, 2002.

[34] J. Jacky, R. Risler, I. Kalet, and P. Wootton. Clinical neutron therapy system control system specification part I. Technical Report 90-12-01, University of Washington, Department of Radiation Oncology, 1990.

[35] J. Jacky, R. Risler, D. Reid, R. Emery, J. Unger, and M. Patrick. A control system for a radiation therapy machine. Technical Report 2001-05-01, University of Washington, Department of Radiation Oncology, 2001.

[36] K. H. John and M. Tiegelkamp. *IEC 61131-3: Programming Industrial Automation Systems Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids*. Springer Publishing Company, Incorporated, 2nd edition, 2010. ISBN 3642120148, 9783642120145.

[37] T. Kelly and R. Weaver. The goal structuring notation – a safety argument notation. In *Proceedings of the Dependable Systems and Networks 2004 Workshop on Assurance Cases*, July 2004.

[38] M. R. Lyu, editor. *Handbook of Software Reliability Engineering*. McGraw-Hill, Inc., 1996.

[39] J. P. Near, A. Milicevic, E. Kang, and D. 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.

[40] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[41] S. Owre, N. Shankar, and J. Rushby. PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction (CADE)*, pages 748–752, Sam Owre, Natarajan Shankar, and John Rushby, June 1992.

[42] J. Rushby. Formalism in safety cases. In *Safety-Critical Systems Symposium*, 2010.

[43] J. Rushby. Mechanized support for assurance case argumentation. In Y. Nakano, K. Satoh, and D. Bekki, editors, *New Frontiers in Artificial Intelligence*, volume 8417 of *Lecture Notes in Computer Science*, pages 304–318. Springer International Publishing, 2014. doi: 10.1007/978-3-319-10061-6_20.

[44] E. Torlak. *A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2009. AAI0821754.

[45] E. Torlak and R. 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.

[46] E. Torlak and R. 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.

[47] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'07, pages 632–647, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-71208-4. URL `http://dl.acm.org/citation.cfm?id=1763507.1763571`.

[48] C. Weinstock, J. Goodenough, and J. Hudak. Dependability cases. Technical Report CMU/SEI-2004-TN-016, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2004. URL `http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=6919`.

[49] Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 2007.