A Lightweight Code Analysis and its Role in Evaluation of a Dependability Case

Joseph P. Near, Aleksandar Milicevic, Eunsuk Kang, Daniel Jackson
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{jnear, aleks, eskang, dnj}@csail.mit.edu

ABSTRACT

A *dependability case* is an explicit, end-to-end argument, based on concrete evidence, that a system satisfies a critical property. We report on a case study constructing a dependability case for the control software of a medical device. The key novelty of our approach is a lightweight code analysis that generates a list of side conditions that correspond to assumptions to be discharged about the code and the environment in which it executes. This represents an unconventional trade-off between, at one extreme, more ambitious analyses that attempt to discharge all conditions automatically (but which cannot even in principle handle environmental assumptions), and at the other, flow- or context-insensitive analyses that require more user involvement. The results of the analysis suggested a variety of ways in which the dependability of the system might be improved.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Design—*Methodologies*

General Terms

Software dependability, safety, reliability

Keywords

Dependability case, problem frames, property-part diagram, code analysis, side conditions

1. INTRODUCTION

The construction of a *dependability case*—an explicit argument that a system satisfies a critical property—is increasingly being advocated not only for assurance (that is, establishing the dependability of a system) [6, 16, 18], but also as an integral part of development [9], by making design and implementation decisions that simplify and strengthen the case and thus improving the dependability of the delivered system.

This paper reports on applying this idea to a medical system—the control software for a proton therapy machine—focusing in particular on the use of a lightweight static analysis. The starting point is the articulation of a safety-critical property of the system; namely, that when the treatment door is inadvertently opened in the middle of a treatment, the system should terminate the delivery of the radiation by inserting a physical beam stop. We constructed a dependability case for the property from the system level down to the code. A static analysis, developed specially for the purpose of this study, was applied to the code, which generated a list of side conditions that represented obligations to be discharged – some about the physical peripherals, and some about the computer infrastructure. The side conditions were classified into those that were more or less reasonable—that is, more or less likely to hold—and based on this classification, some design and implementation alterations were proposed that might strengthen the dependability case.

The contributions of the paper include:

- A concrete illustration of an approach to constructing a dependability case that we have outlined in earlier papers [14, 15] involving: (1) identification of partial but critical properties of the problem domain, (2) construction of a *property-part diagram* [10] showing the relationship between component specifications and system-level requirements, and (3) checking of the code against the component specifications.

- A lightweight code analysis based on symbolic execution that (1) incorporates user-provided specifications so that, in particular, calls to a middleware API can be interpreted appropriately, (2) discharges control flow conditions automatically using symbolic state information, but delegates to the user the harder problem of discharging conditions that are related to the state of external peripherals, and (3) presents the results to the user for evaluation in a form that aids navigation and review.

- A strategy for improving the dependability of the system by generating a list of side conditions that correspond to environmental assumptions, reviewing them manually, and then adjusting the design and implementation to eliminate the less desirable assumptions.

The paper is organized as follows. We begin by outlining the context of the case study: the basic structure of the installation and the software, and the existing safety mechanisms (including a redundant hardware layer) that mitigate potential safety problems (Section 2). We discuss the critical property considered, and show how property-part diagrams were constructed for them (Section 3). We then describe the code analysis that we performed (Section 4), and the results of the analysis (Section 5): the list of conditions that were generated, and how they were evaluated. We discuss the recommendations that were made to the developer of the system to improve the safety of the system, and lessons that we learned from the case study (Section 6). We conclude with a discussion of related work (Section 7).

2. OVERVIEW OF THE SYSTEM

The Burr Proton Therapy Center (BPTC) at the Massachusetts General Hospital is one of only a handful of places in the world offering radiation therapy with protons. In contrast to gamma ray and electron therapy, proton therapy offers more precise targeting and therefore less collateral damage, making it safer and more effective for a variety of conditions, in particular tumors of the eye, brain tumors, and tumors in small children.

Like most other software-controlled medical devices, the proton therapy system at the BPTC is complex. There are three treatment rooms, each containing a robotic positioning couch on which the patient lies, a gantry allowing coarse aiming of the proton beam, and a nozzle for fine control over the beam. The beam from the cyclotron is multiplexed between the rooms, and guided to the nozzles by a system of electromagnets. In the basic treatment scenario, the therapist calls up the prescription from a database, positions the patient and the gantry, and submits a beam request. Technicians in the master control room oversee the cyclotron and system settings, and grant requests by allocating the beam.

Since 2003, the Software Design Group at MIT has collaborated with the development team at the BPTC, led by the center's director Dr. Jay Flanz, on a project exploring new ways to structure and analyze software to ensure safety while retaining flexibility to extend functionality. The BPTC team is currently working on a major enhancement of the system that will allow "pencil beam scanning", in which a fine beam scans the tumor, in contrast to the current approach, which uses a more diffuse beam whose outline is contained by custom collimators that are inserted in the nozzle aperture.

The BPTC system incorporates a number of safety mechanisms. The beam emerges from the nozzle and passes through an ionization chamber allowing a redundant dose check before arriving at the patient. A completely separate hardware system based on programmable logic arrays and independent hardware relays checks for a variety of critical conditions, and in particular can shut off the beam if the preset overdose is reached or exceeded.

In addition, the system contains various safety features to mitigate the risk of inadvertent radiation exposure not only to patients, but also to employees of the hospital (i.e. therapists and physicians). For this case study, we chose the critical property of one particular safety feature, and constructed a dependability case to argue that the system satisfies the property:

Door safety property: When the treatment door is opened in the middle of a treatment session, the system inserts the beam stop to terminate the treatment.

In a previous study, we had investigated a similar property using an ad hoc approach, based on a mostly manual review of the code [22]. In this study, we analyzed the door safety property using the framework we have recently developed [14, 15] and using a new code analysis tool built for this purpose.

3. DEPENDABILITY CASE

A *dependability case* is an explicit, end-to-end argument that a system satisfies a critical property. It is *explicit*, in that it provides concrete *evidence* that the system establishes the property, and *end-to-end*, that its claim about dependability spans the system both horizontally (in terms of real-world phenomena of interest, from input to output) and vertically (from the design level down to the code). These two characteristics contrast with the conventional *process-based* approach to dependability, which typically mandates a particular regimen of documentation, testing (and perhaps even formal verification), and does not require concrete evidence that these efforts ensure dependability.

3.1 Overview of Construction

In our approach, the construction of a dependability case involves the following steps:

1. **Identifying critical system-level properties:** A large, complex system usually has a long list of requirements, some of which are more critical to dependability than others. The first task in constructing a case involves prioritizing the requirements and identifying the most critical ones.

2. **Articulating the system structure:** Having identified a critical property, the parts of the system and their interactions are described using a *problem diagram* from Jackson's *problem frames* approach [13]. A hallmark of this approach is the distinction between parts that are components of the *machine* to be built (e.g., modules of the software) and parts that are *domains* in the environment (e.g., physical peripherals or human operators). This distinction is essential to a dependability case, since the critical requirements that are of interest are stated in terms of phenomena in the real world, which are often not observable at the interface of the machine. For example, the requirement that the door opening causes the beam stop to be inserted concerns the domains in which the door and the beam stop are situated; how the software behaves at its interface with these domains will, of course, turn out to be vital, but is not relevant at the requirements level.

3. **Assigning properties to parts:** The critical requirement is decomposed into properties on individual parts of the system. A property on a software component is the specification that the software must be implemented to fulfill; a property on an environmental domain is an assumption about how the domain behaves. We use a *property-part diagram* [10] to express dependencies between properties and parts. The diagram illustrates how different parts of the system, each satisfying its own property, together establish the high-level requirement. The resulting dependency structure in the property-part diagram corresponds to the design-level argument in the dependability case.

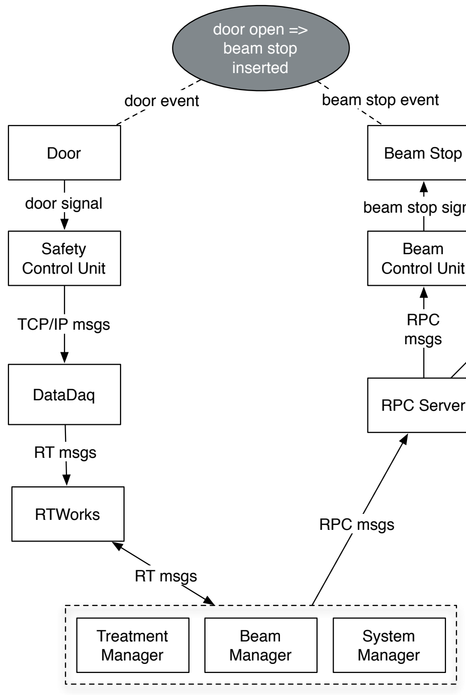


Figure 1: Problem diagram for the door safety property. A box represents a system part, and the label on an edge between two boxes a shared phenomenon through which they interact. Related parts are grouped inside a dotted box. The requirement (the circle) is expressed in terms of phenomena on two environmental domains, the door and the beam stop.

4. **Analyzing the design-level case:** The property-part diagram is analyzed to ensure that the decomposition from the previous step is correct: namely that the combination of the properties on the individual parts together satisfy the system requirement. The analysis may help reveal a missing domain assumption or a weak specification.

5. **Checking each individual part:** Finally, after having established the dependability case at the design-level, the implementation of each software component is checked to ensure that it conforms to its specification. Discharging a domain assumption involves consulting a domain expert to ensure that the assumption is a reasonable characterization of the domain's behavior.

3.2 Building a Case for the Door Safety

We demonstrate our approach by describing the construction of a dependability case for the door safety property in the BPTC system. Our goal is to build a case to argue that when the door is opened during treatment, the system inserts the beam stop to immediately halt the delivery of radiation.

The problem diagram in Figure 1 illustrates the structure of the BPTC system, which consists of: (1) physical equipment, such as the treatment door and the beam stop, (2) low-level *control units*, which interact with hardware sensors and actuators to control the physical equipment, (3) high-level *managers*, which direct the control units to carry out the treatment, and (4) intermediate *communication mechanisms*, including standard TCP/IP and RPC protocols, as well as an off-the-shelf messaging library called *RTworks* [25]. The software components run on commodity UNIX workstations on a local area network. The software consists of around 250,000 lines of C code, distributed across several hundred source files.

The system parts interact with each other by sharing *phenomena*, which include, for example, signals generated by a physical device and sensed by a control unit, and RTworks and RPC messages sent over the network. In the diagram, these phenomena appear as labels on the edges between the parts; the arrow on an edge shows the direction in which information associated with a phenomenon flows from one part to another.

In a typical scenario, the parts of the BPTC system communicate with each other in the following manner. A control unit, initiated by a hardware signal, sends a TCP/IP message to an intermediate component called *DataDaq*. DataDaq then relays the message to a manager that is responsible for handling the event through RTworks. The manager processes the message and prepares an appropriate response, by delegating the task to another manager through RTworks, or by sending an RPC message directly to a control unit.

An important aspect of the problem diagram is that it states the door safety property *only* in terms of phenomena that occur in the environment—namely, the events of the door opening and the beam stop being inserted. It does not mention anything about phenomena inside the software components (i.e. the control units, DataDaq, or the managers). Thus, our next task involves using the problem diagram as a starting point to derive domain assumptions and specifications that are sufficient to establish the property, and assign them to the corresponding parts.

The problem diagram is elaborated into a property-part diagram (Figure 2) that shows how the requirement is decomposed into properties on the individual parts of the system. In this case, all the properties take a simple form—that one event leads to another—and are stated informally. An edge in the property-part diagram shows how one property is discharged either by a combination of other properties, or by a component of the system.

For example, the property on DataDaq states that "when it receives a message from the Safety Control Unit, indicating that the door has been opened, DataDaq notifies the Treatment Manager by invoking the function rtdaqinDoorOpen"; this is a partial specification that must be fulfilled by the implementation of DataDaq for the door safety property to hold. When the Treatment Manager is notified of the door opening, it instructs the Beam Control Unit to insert the beam stop, eventually by invoking the function bsInsert; this property is shown in the diagram as being fulfilled together by the two managers. Since DataDaq uses RTworks to communicate to the managers, the specification of DataDaq depends on the correct behavior of RTworks in handling requests to create and send messages. Hence, there is an edge from the property of DataDaq to that of RTworks. Similarly, the managers rely on RTworks and the RPC server for communication to fulfill their responsibility.

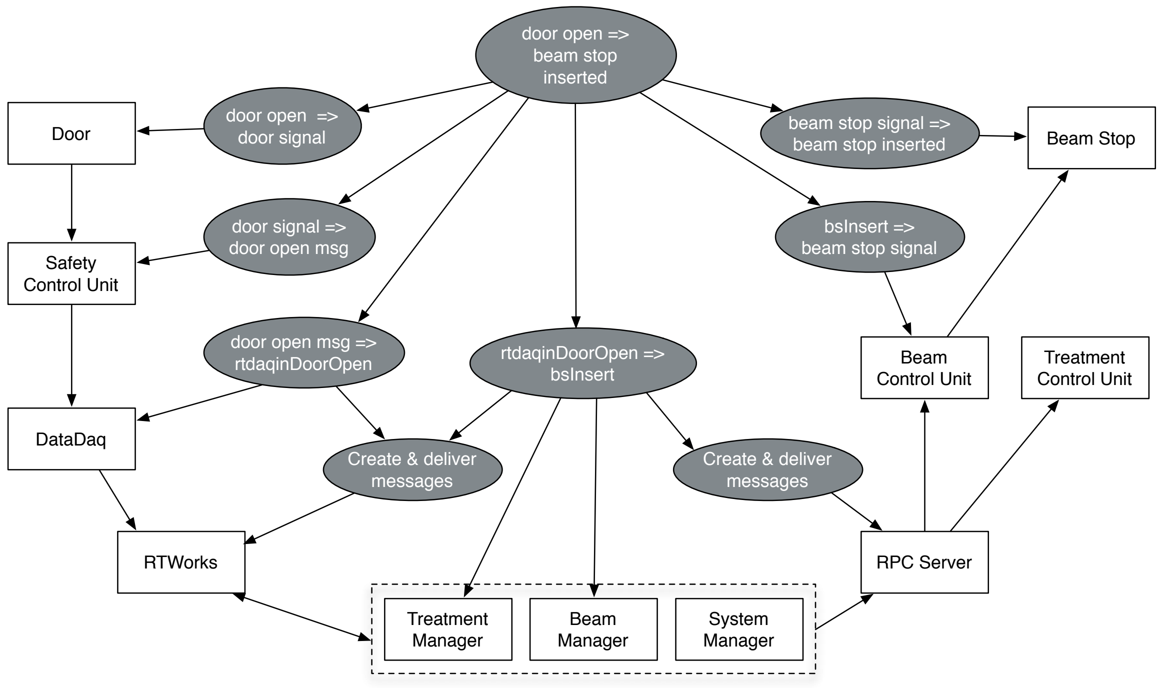

door open => beam stop inserted
door open => door signal
Door
beam stop signal => beam stop inserted
Beam Stop
door signal => door open msg
Safety Control Unit
bsInsert => beam stop signal
door open msg => rtdaqinDoorOpen
rtdaqinDoorOpen => bsInsert
Beam Control Unit
Treatment Control Unit
DataDaq
Create & deliver messages
Create & deliver messages
RTWorks
RPC Server
Treatment Manager
Beam Manager
System Manager

Figure 2: Property-part diagram for the door safety property. A box represents a part, and a circle represents a property. An edge originating from a circle shows a dependency of the denoted property on a part, or another property. An edge between two boxes shows interaction between the parts (for simplification, we omit the labels that represent shared phenomena.).

The door safety property depends not only on the specifications of software, but also on assumptions on the environment. When the treatment door is opened, we expect the door sensor to generate a signal to be sent to the Safety Control Unit. If the sensor fails to do so for some reason (e.g. a power failure), then the system will not insert the beam stop, regardless of how the rest of the system behaves. Thus, stating these domain assumptions is crucial to achieving dependability.

The dependency structure in the property-part diagram corresponds to the argument that the system establishes the door safety property. If the software components satisfy their specifications, and the domains behave as expected, then the system should insert the beam stop when the door is opened. The properties on the individual parts correspond to the premises of the argument; these premises must be shown to hold by analyzing the implementation or discharging the domain assumptions. The argument, together with evidence for discharging its premises, forms the dependability case for the door safety property. In Section 4, we describe the code analysis that we performed to produce evidence that the implementation of the Treatment and Beam Managers conform to their specifications.

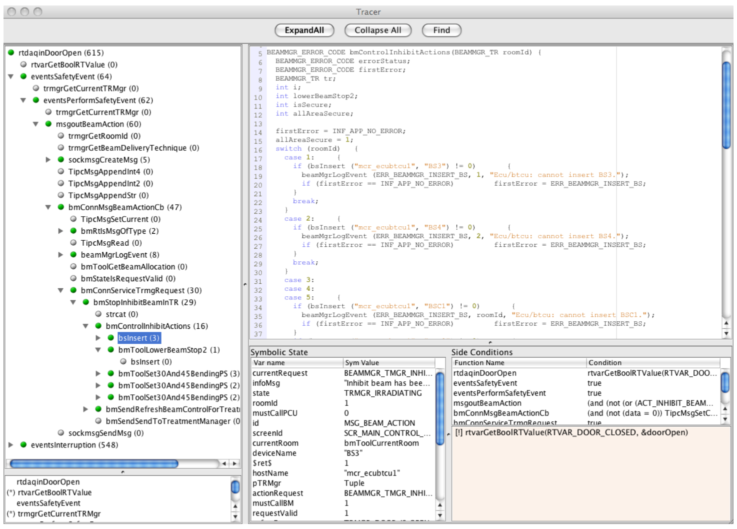
If properties and parts are specified formally, an argument can be mechanically checked for validity; that is, the combination of the specifications and domain assumptions indeed establish a critical property. We modeled the BPTC system in the Alloy modeling language [8], and analyzed the argument for the door safety property using the Alloy Analyzer, a SAT-based analysis tool. Due to limited space, we do not describe the model in this paper; however, the complete Alloy model is available on `http://people.csail.mit.edu/eskang/mgh-model`.

4. CODE ANALYSIS

As noted above, in this particular case, the properties asserted of the software components in the property-part diagram all take a particular form: that the occurrence of one event leads to another. In the code, these properties translate into a relationship in the call graph: that calling one function inevitably leads to calling another. The property on the managers that support the door safety property, for example, is rtdaqinDoorOpen => bsInsert; rtdaqinDoorOpen is a function in the implementation of the Treatment Manager, and bsInsert is a function in the Beam Control Unit. Having formulated the desired properties of the software this way, a lightweight analysis is used to show that a call to rtdaqinDoorOpen really does cause the call to bsInsert.

The analysis takes a component and its partial specification, and produces a set of *side conditions* that, if true, imply that the component satisfies its specification. If the analysis produces no side conditions, then the analysis has proved the component correct with respect to the specification. More often, however, the analysis fails to discharge all of the side conditions, and leaves some for the user to manually discharge.

The analysis scales to large codebases for two reasons. First, it examines only a single component at a time—an approach that is made possible by the practice of decomposing requirements into partial specifications on individual parts. Second, the analysis is designed to give up quickly. As the analysis navigates the path from origin to destination,



```
rtdaqinDoorOpen (615)
  rtvarGetBoolRTValue (0)
▼ eventsSafetyEvent (64)
    trmgrGetCurrentTRMgr (0)
  ▼ eventsPerformSafetyEvent (62)
      trmgrGetCurrentTRMgr (0)
    ▼ msgoutBeamAction (60)
        trmgrGetRoomId (0)
        trmgrGetBeamDeliveryTechnique (0)
      ▶ sockmsgCreateMsg (5)
        TipcMsgAppendInt4 (0)
        TipcMsgAppendInt2 (0)
        TipcMsgAppendStr (0)
      ▼ bmConnMsgBeamActionCb (47)
          TipcMsgSetCurrent (0)
        ▶ bmRtIsMsgOfType (2)
          TipcMsgRead (0)
        ▶ beamMgrLogEvent (8)
          bmToolGetBeamAllocation (0)
          bmStateIsRequestValid (0)
        ▼ bmConnServiceTrmgRequest (30)
          ▼ bmStopInhibitBeamInTR (29)
              strcat (0)
            ▼ bmControlInhibitActions (16)
              ▶ bsInsert (3)
              ▼ bmToolLowerBeamStop2 (1)
                  bsInsert (0)
              ▶ bmToolSet30And45BendingPS (3)
              ▶ bmToolSet30And45BendingPS (2)
              ▶ bmToolSet30And45BendingPS (2)
            ▶ bmSendRefreshBeamControlForTreat
              bmSendSendToTreatmentManager (0
        sockmsgSendMsg (0)
▶ eventsInterruption (548)
```

```
    rtdaqinDoorOpen
(*) rtvarGetBoolRTValue
    eventsSafetyEvent
(*) trmgrGetCurrentTRMgr
```

```
BEAMMGR_ERROR_CODE bmControlInhibitActions(BEAMMGR_TR roomId) {
  BEAMMGR_ERROR_CODE errorStatus;
  BEAMMGR_ERROR_CODE firstError;
  BEAMMGR_TR tr;
  int i;
  int lowerBeamStop2;
  int isSecure;
  int allAreaSecure;

  firstError = INF_APP_NO_ERROR;
  allAreaSecure = 1;
  switch (roomId)   {
    case 1:     {
      if (bsInsert ("mcr_ecubtcu1", "BS3") != 0)        {
        beamMgrLogEvent (ERR_BEAMMGR_INSERT_BS, 1, "Ecu/btcu: cannot insert BS3.");
        if (firstError == INF_APP_NO_ERROR)         firstError = ERR_BEAMMGR_INSERT_BS;
      }
      break;
    }
    case 2:     {
      if (bsInsert ("mcr_ecubtcu1", "BS4") != 0)        {
        beamMgrLogEvent (ERR_BEAMMGR_INSERT_BS, 2, "Ecu/btcu: cannot insert BS4.");
        if (firstError == INF_APP_NO_ERROR)         firstError = ERR_BEAMMGR_INSERT_BS;
      }
      break;
    }
    case 3:
    case 4:
    case 5:     {
      if (bsInsert ("mcr_ecubtcu1", "BSC1") != 0)        {
        beamMgrLogEvent (ERR_BEAMMGR_INSERT_BS, roomId, "Ecu/btcu: cannot insert BSC1.");
        if (firstError == INF_APP_NO_ERROR)         firstError = ERR_BEAMMGR_INSERT_BS;
      }
```

Symbolic State

Var name	Sym Value
currentRequest	BEAMMGR_TMGR_INHI...
infoMsg	"Inhibit beam has bee...
state	TRMGR_IRRADIATING
roomId	1
mustCallPCU	0
id	MSG_BEAM_ACTION
screenId	SCR_MAIN_CONTROL_...
currentRoom	bmToolCurrentRoom
deviceName	"BS3"
ret	1
hostName	"mcr_ecubtcu1"
pTRMgr	Tuple
actionRequest	BEAMMGR_TMGR_INHI...
mustCallBM	1
requestValid	1

Side Conditions

Function Name	Condition
rtdaqinDoorOpen	rtvarGetBoolRTValue(RTVAR_DOO...
eventsSafetyEvent	true
eventsPerformSafetyEvent	true
msgoutBeamAction	(and (not (or (ACT_INHIBIT_BEAM...
bmConnMsgBeamActionCb	(and (not (data = 0)) TipcMsgSetC...
bmConnServiceTrmgRequest	true

[!] rtvarGetBoolRTValue(RTVAR_DOOR_CLOSED, &doorOpen)

Figure 3: Snapshot of the prototype tool displaying a list of side conditions to reach bsInsert.

it attempts to evaluate conditional expressions using symbolic state, but the set of conditions that can be evaluated without user input is small. To mitigate this, the analysis prompts the user for more information (for example when a procedure whose code is missing is called), and if it cannot evaluate a condition at all, adds it to the list of generated side conditions to be reported to the user and discharged by other means.

This lightweight style of code analysis is intended to support a feedback loop between the design and implementation steps. A side condition represents a proof obligation, but it also gives the user information about how the design of the system can be improved, since a side condition that is too difficult for the analysis to handle often points to potential design flaws, such as tight coupling and unnecessary dependencies. After examining the set of side conditions the analysis generates, the user may choose to redesign the system to eliminate the most problematic of these conditions, update the implementation, and rerun the analysis.

A screenshot of our tool appears in Figure 3. The left-hand pane contains the tree of function calls leading from the premise to the conclusion of the desired property, the top-right-hand pane allows the user to browse the code, and the bottom-right-hand pane contains a summary of the symbolic environment used in the analysis and the side conditions the analyzer has discovered.

4.1 Strategy and Implementation

To check whether components in the BPTC system propagate events as they should, we implemented a lightweight, flow-sensitive, context-sensitive, static analysis. Only a single path need be followed, since—in this particular system—all branches away from the normal path represent erroneous behavior. This makes the analysis very lightweight, since it avoids the path-explosion problem entirely. It is still necessary, however, to ensure that this normal path is indeed followed, and that requires evaluating the conditions along the way.

The evaluation of conditions is based on abstract state computed by the analysis. The state obtained by the symbolic evaluation alone is not, however, sufficient to evaluate all conditions. In addition, the analysis needs *domain knowledge* that constrains the initial state, indicates what values are returned by calls to missing code, and gives the source and destinations of messages (which are determined, in the implementation, by a publish-subscribe subsystem that registers event types with handlers using initialization files in a special language). This domain knowledge is encoded in tables provided by the user. These tables were constructed from the specification documents of the system, and using the domain knowledge of the developers.

4.2 Analysis Algorithm

The pseudocode describing the analysis is outlined in Figure 4. For each statement in the code's control flow graph, the top-level analysis function Analyze applies the function AnalyzeStmt with the current symbolic environment env. The algorithm has global access to the entire codebase through the data structure code, which maps each function to a list of statements. It also has access to domain_knowledge, which contains the set of user-provided information about the symbolic state.

The analysis proceeds as follows. When the statement to be analyzed is a variable assignment, we simply assign the symbolic value of the right-hand-side expression to the environment (line 12). If the statement is a conditional, we first attempt to determine whether or not the condition can be resolved to a constant (lines 15-16). If not, then we check the two branches to see if any one of them will immediately return an error. If so, we add the condition that would lead to the non-erroneous branch as a side condition (using the helper function GenerateSideCondition), and analyze that branch (lines 22-27). If we cannot immediately determine which branch to proceed into, then we pause the analysis, and ask the user for more domain knowledge to resolve the condition (line 30). Once the symbolic environment has been updated with a new piece of domain knowledge, we re-analyze the statement.

If the statement is a function call, and the code for the function is available, then we simply analyze the body of the code (line 35). If not, then we first check whether the information about the function call already exists in the cur-

```
 1 /* Input: a list of statements to be analyzed, and
 2         symbolic environment */
 3 Analyze(stmts, env):
 4   for each stmt in stmts:
 5     AnalyzeStmt(stmt, env)
 6
 7 /* Input: statement to be analyzed, and
 8         symbolic environment */
 9 AnalyzeStmt(stmt, env):
10   if stmt = ‘‘v := e’’ then
11     /* variable assignment */
12     env(v) := eval(AnalyzeStmt(e), env)
13   else if stmt = ‘‘if(cond) then b1 else b2’’ then
14     /* conditional */
15     AnalyzeStmt(cond)
16     switch eval(cond, env)
17       case TRUE:
18         Analyze(b1, env)
19       case FALSE:
20         Analyze(b2, env)
21       case SYMBOLIC:
22         if b1 is ‘‘error block’’ then
23           GenerateSideCondition(!cond)
24           Analyze(b2, env)
25         else if b2 is ‘‘error block’’ then
26           GenerateSideCondition(cond)
27           Analyze(b1, env)
28         else
29           /* can’t proceed; ask user */
30           Pause()
31   else if stmt = ‘‘f(a, b, ...)’’ then
32     /* function call */
33     if f in code then
34       /* code for f available; analyze it */
35       Analyze(code[f(a, b, ...)], env)
36     else if ‘‘f(a, b, ...)’’ in domain_knowledge then
37       /* return value of f is known; update environment
             */
38       Update(env, domain_knowledge[f(a, b, ...)])
39       GenerateSideCondition(‘‘f(a,b,..) does not crash’’)
40     else
41       /* can’t proceed; ask user */
42       Pause()
```

Figure 4: Pseudocode for the analysis algorithm.

rent set of domain knowledge. If so, we update the symbolic environment accordingly, and generate a side condition that says that the function must not crash, since this would lead to an undesirable consequence of the program never reaching its goal (lines 38-9). Finally, if no information is known about the missing function, we prompt the user for domain knowledge (line 42).

If the analysis completes successfully but does not reach its goal (in our case, bsInsert, the procedure that inserts the beam stop), then this implies either that the implementation of the system is incorrect (i.e. does not conform to the specification), or that the domain knowledge provided by the user is insufficient.

In general, the analysis prompts the user for precisely the domain knowledge it needs. It is designed to be an interactive process between the user and the analyzer; when the analysis fails, the user inspects the reason for failure and adds domain knowledge to avoid that problem. In practice, this design leads to specific questions to be posed to domain experts. Domain knowledge therefore involves no guesswork—if the domain expert can answer the question, then the correct domain knowledge has been obtained.

```
 1 STATIC BOOLEAN rtdaqinDoorOpen(RTVAR_ID id,
 2                              TRMGR_STATE state) {
 3
 4  BOOLEAN doorOpen, areaNotSecured,
 5          keyInServiceMode, tcrKeySwitch;
 6  if (state==TRMGR_INITIALIZATION) {
 7    return TRUE;
 8  }
 9  if (!rtvarGetBoolRTValue(id, &doorOpen)) {
10    TRACE_ERROR_MSG();
11    return FALSE;
12  }
13
14  if (doorOpen) {
15   if (!eventsSafetyEvent(TRMGR_DOOR_IS_OPEN)) {
16      TRACE_ERROR_MSG();
17      return FALSE;
18   }
19   if ((state==TRMGR_IRRADIATING) ||
20     (state==TRMGR_MANAGING_INTERRUPTION)) {
21      if (!eventsInterruption(TRMGR_DOOR_IS_OPEN)){
22        TRACE_ERROR_MSG();
23        return FALSE;
24      }
25    }
26  } else {
27    ...
28  }
29  return TRUE; }
```

Figure 5: Example C code from the Treatment Manager software.

4.3 Example

As an example of our analysis, consider the code snippet in Figure 5: the first part of rtdaqinDoorOpen. The analysis proceeds as follows:

1. The first conditional (line 6) is encountered, and the analysis is unable to determine which branch to follow. The user must provide domain knowledge stating that state != TRMGR_INITIALIZATION.

2. The second conditional (line 9) is encountered, and the analysis determines that the false branch must be chosen in order to avoid an error. It emits the side condition rtvarGetBoolRTValue(id, &doorOpen)= TRUE.

3. The third conditional (line 14) is encountered, and the analysis is unable to determine which branch to follow. The user must add domain knowledge stating that rtvarGetBoolRTValue(id, &doorOpen) sets doorOpen to TRUE.

4. The fourth conditional (line 15) is encountered, and the analysis examines the eventsSafetyEvent function and determines that it returns TRUE.

5. The fifth conditional (lines 19-20) is encountered, and the analysis is unable to determine which branch to follow. The user must add domain knowledge that state = TRMGR_IRRADIATING (in other words that the system is in the irradiating state).

6. The sixth conditional (line 21) is encountered, and the analysis finds the call to eventsInterruption. The analysis generates the side condition that eventsInterruption(TRMGR_DOOR_IS_OPEN) returns TRUE (that is, that the treatment room door is indeed open).

When the analysis is complete, then, the user has provided the following domain knowledge:

- state != TRMGR_INITIALIZATION (the treatment manager is not in the state corresponding to initialization).
- rtvarGetBoolRTValue(id, &doorOpen) sets doorOpen to TRUE (that the door is open).
- state = TRMGR_IRRADIATING (the treatment manager is in the irradiating state).

and must manually discharge the following side conditions:

- rtvarGetBoolRTValue(id, &doorOpen)= TRUE (that the library call for determining whether the door is open completes successfully).
- eventsInterruption(TRMGR_DOOR_IS_OPEN)= TRUE.

4.4 Analysis Effort

Since our analysis involves interactions with the user to gather domain knowledge, we did not measure the elapsed time for the analysis. After all of the domain knowledge has been provided by the user, the analysis took 45 seconds to generate all of the side conditions on an Intel Dual Core CPU (2.93 GHz) running Ubuntu. Even though the entire BPTC system consists of 250,000 lines of code across hundreds of C files, once the number of paths have been narrowed down to the single critical path using the domain knowledge, the analysis had to explore only 1026 lines of code across 17 source files.

We consulted one of the developers of the BPTC system, who provided us with the necessary knowledge about the system state at the time when rtdaqinDoorOpen is called, and the expected behavior of the function calls in the managers. The number of pieces of domain knowledge required for this analysis were relatively small (12), and mainly involved the API calls to RTworks and the RPC server. This illustrates the appealing feature of our analysis; by utilizing knowledge of a human expert to discharge side conditions, we were able to avoid analyzing large portions of the codebase (if available) that would have normally been tackled by a more conventional analysis.

The BPTC system is large enough that manual inspection of the code—or even the code of a single component—would be nearly impossible. Our lightweight analysis allows the inspector to direct his or her attention to the lines of code that are relevant to the property being investigated, making manual inspection tractable, even for very large pieces of software.

(1) Event Logging
evtRepReportEvent()= TRUE
evtRepForwardEvent()= TRUE
evtRepSendMessage()= TRUE
hciLoggerLog()= TRUE

(2) System Calls
sprintf(strBuffer, "%s %s", hostName, deviceName)
strncpy(shortTextEvent, newPartTextEvent, 120)
strlen(newPartTextEvent)
fprintf(hciLogFile, s)
strcat(infoMsg, "TR 1.")
(all calls must not cause a segmentation fault)

(3) Message Building
ACT_INHIBIT_BEAM >= ACT_FIRST_ACTION
ACT_INHIBIT_BEAM <= ACT_LAST_ACTION
TipcMsgAppendInt4(beamActionMsg, ACT_INHIBIT_BEAM)= TRUE
TipcMsgAppendInt2(beamActionMsg, 1)= TRUE
TipcMsgAppendInt2(beamActionMsg, DOUBLE_SCATTERING_MODE)= TRUE
TipcMsgAppendStr(beamActionMsg, "")= TRUE

(4) Global State & External Domains
rtvarGetBoolRTValue(RTVAR_DOOR_CLOSED, &doorOpen)= TRUE
trmgrGetCurrentTRMgr = *current treatment manager*
trmgrGetRoomId = *current room ID*
trmgrGetBeamDeliveryTechnique()= DOUBLE_SCATTERING_MODE
bmToolGetBeamAllocation()= TRUE
bmStateIsRequestValid()= TRUE

Figure 6: Side conditions for the door safety property, categorized in terms of their characteristics.

5. RESULTS

Figure 6 shows the results of the analysis, as a list of 20 side conditions, sorted by our determination of their type.

First, we found that the safety-critical code we analyzed depends on less critical components that perform logging (Figure 6 (1)). The three evtRep functions report events to the logging facilities, while the hciLoggerLog sends events to the HCI manager to display them to the user. Both of these logging components depend on system calls for formatting strings and writing them to files on disk (Figure 6 (2)). Operations like strcat, sprintf, and strncpy can cause segmentation faults; fprintf can fail if the filesystem is full. Either situation might cause the beam stop not to be inserted.

Second, the process for building and invoking an RPC call takes several steps, and all must complete successfully for the beam stop to be inserted (Figure 6 (3)). The first two side conditions in this section represent properties of message types that can easily be verified manually; the others, however, rely on the proprietary Smart Sockets implementation of the RPC server.

Finally, we discovered many side conditions related to the use of global state (Figure 6 (4)). The rtvarGetBoolRTValue function, for example, looks up the value of a global variable; the trmgrGetCurrentTRMgr function inspects the global state to obtain a pointer to the current Treatment Manager. It is difficult to build confidence in code that depends on global state because less critical code could modify that state at any time. If the global state is not in the expected configuration, the safety-critical code may not function correctly. This dependence means that *all* code with access to the global state must be treated as safety-critical, and analyzed as such; this task was outside the scope of our case study.

6. DISCUSSION

6.1 Recommendations

We found no evidence of bugs in the implementation. Moreover, a more sophisticated analysis might have been able to discharge some of the side conditions reported (but still excluding the side conditions that reflect properties of the environment).

In our view, however, we treat the undischarged side-conditions as suggestions for how the system might be improved so that its safety is more evident. In this section, we review these suggestions. Some are easily implementable with few changes to the existing codebase; others would require more significant redesign of the system. (It should be noted that the software is backed by a redundant hardware safety system, so the software is not a single point of failure.)

First, some of the side conditions can be eliminated by replacing them with ones that can be more easily discharged, or by reordering parts of the implementation. For example, string manipulation might be done using a safe string library [19] instead of standard C functions; this would eliminate the side conditions on system calls. Error logging might be done after instead of before the beam stop is inserted, eliminating the side conditions related to logging. Both of these examples involve relatively straightforward refactoring, and would strengthen the dependability case.

Other side conditions point to larger issues in the design of the system, and may suggest changes at the architectural level. Highly critical functionality (e.g., the response to the door opening) depends on generic components (e.g., RTworks) that serves multiple areas of functionality, some of which are of lower priority (e.g., patient positioning). A better approach might be to construct a separate, reliable communication pathway reserved for emergency functions (such as the beam stop insertion), thereby eliminating the dependency on RTworks.

6.2 Lessons Learned

Theorem provers and static analyzers are designed to *prove* that a system satisfies a property—they do not leave behind side conditions for the user to discharge. In our experience with the BPTC system, producing a complete proof would be very difficult, because the set of components involved in producing the proof is large, and includes components for which neither code not precise specifications are available. By introducing a human element into the analysis effort, and by dealing with difficult proof obligations by eliminating them from the critical path rather than by proving them correct, we are able to balance investment in analysis and design, and obtain some degree of confidence from more lightweight analyses.

The way a system is structured and coded has a big influence on how effective such a lightweight analysis can be (as noted by Griswold [7]). In this study, the systematic structure of the BPTC system was a great help:

- Because of the pervasive use of events (rather than, for example, communication through global variables), it was easy to formulate properties in terms of simple control flow. Many of the specifications in the BPTC system had the form "if event A happens, then event B should happen." This makes the analysis easier to implement; with some domain knowledge about which functions send and receive events, the analysis can check code against such a specification without the use of a constraint solver. Even specifications that do not naturally take this form can often be recast in terms of the events that cause the desired outcome.

- Because event causality is transitive, there is a natural modularity that the analysis exploits; for each module, the analysis connects an originating event to a target event.

- Consistent coding conventions can be exploited in the analysis. The BPTC code, for example, uses the convention that each function call returns `TRUE` for success, and `FALSE` for failure. The programmers also used a standard set of functions to indicate error conditions. These conventions can be encoded as domain knowledge, and can be used by the analysis to improve both scalability and precision. For example, by expecting most functions to return `TRUE`, our analysis is able to emit more specific side conditions for missing functions.

6.3 Limitations

Some analyses are difficult to perform using our lightweight engine. Our analysis cannot handle extensive global state, dynamic allocation or concurrency, and does not address timing constraints. Additionally, the soundness of our analysis is dependent upon the accuracy of the domain knowledge the user provides about environmental domains. The analysis offers no guidance in discharging the domain knowledge; instead, it relies on the user to consult with a domain expert—for example, a physicist who understands the expected behavior of the proton beam, or a technician with the knowledge of the beam stop mechanics.

Our analysis by itself does not provide sufficient evidence to support a dependability case. To further strengthen the case, it should be combined with other techniques, including extensive unit and integration testing, historical data on the reliability of framework components, more sophisticated and complete analysis of critical components, and evaluation by domain experts of domain assumptions. Unifying results from different kinds of analyses to demonstrate dependability remains a challenging research problem [4].

7. RELATED WORK

Dependability Cases Traditional approaches to evaluating dependability of software systems in industry are *process-based*. In these approaches, a system is considered suitable for certification by a government agency if its development adheres to one or more standards, such as Common Criteria [2] or IEC 61508 [23]. The main criticism of process-based approaches is the lack of an evident link between the extent of the quality assurance activities that are mandated by the standards and the level of dependability that is inferred [11].

In response, a number of case-based approaches to software dependability have been developed. The common goal behind these approaches is to provide an argument that directly links the developer's claims about the dependability of the system to concrete evidence (e.g. testing reports, formal proofs, etc.) that support them. Our approach belongs to this group.

Kelly introduced the *Goal-Structuring Notation* (GSN) as a way to represent a *safety case*—an argument that the system satisfies its safety properties [16]. Assurance based development (ABD) [6] is a methodology that integrates an *assurance case* with the development of a dependable system[1]. In this approach, based on the GSN, the top system-level requirements are decomposed into smaller subgoals, which are discharged using various *strategies.* Lutz and Patterson-Hine [17] proposed an approach to building an argument to support the system's ability to detect and handle safety-related contingencies. Like ABD, the structure of their argument is based on the GSN, but they discharge subgoals using analysis of fault models.

In both of these approaches, the premises of an argument (sub-properties that together imply the system-level property) are linked to the *methods* by which they are discharged; these methods may include testing, inspection, or formal methods. In contrast, the property-part diagram connects the premises (the specifications and domain assumptions) to the *parts* that are responsible for fulfilling them. In other words, whereas these approaches tend to focus on the structure of the development process for evidence of dependability, we instead focus on the structure of the product.

Code Analysis There is a rich body of work in the literature on techniques for checking code conformance against specifications. Many of them deal with checking behavioral specifications or heap invariants [1, 3, 5, 12, 26]. Our approach is much more lightweight in nature: (1) it does not attempt to discharge the side conditions itself, and (2) it relies on the user's domain knowledge to prune the search space down to a single critical path.

In this respect, our work is similar to Murphy, Notkin, and Sullivan's *reflexion model* [20]. The user of the reflexion model tool provides a mapping between portions of the source code and their counterparts in the high-level system model, similar to the way in which our approach relies on the user's domain knowledge. The reflexion model is concerned with *system-wide* structural conformance of the code against the design, while our analysis is applied to ensure a behavioral property at the individual component level.

[1]We prefer the more general term "dependability case" over the more common terms "safety case", which seems to exclude systems that are mission-critical but not safety-critical, and "assurance case", which de-emphasizes the role of cases in design.

Giving special treatment to events, and informing the analysis of those events through specifications, is not a new idea, and is part of the framework by Popescu, Garcia, and Medvidovic [21]. By extending the idea of domain knowledge to function specifications, we can also analyze functions for which we have no implementation. Snelting and his colleagues describe an analysis technique to extract a set of conditions that need to hold for a path to be taken between two locations in a program (called *path conditions*), and use a constraint solver to find an input value that could lead to safety violation [24]. Although our side conditions are similar to theirs in nature, it would be difficult to generate many of the conditions that arise in the BPTC system automatically using their technique (for example, a condition on a function for which the source code is not available, such as RTworks).

8. CONCLUSION

Most published efforts to establish the safety of critical systems have taken the system as a given, and have focused on the question of what analysis is sufficient to discharge all proof obligations. In our approach, we seek a balance between analysis and design, trying to reduce the critical path that is responsible for fulfilling critical properties, so that a simpler analysis can be used to establish them. The analysis we developed for this case study supports this balance by generating side conditions that are either discharged by manual analysis, or eliminated by refactoring. In applying this analysis to the therapy system, we found that the set of side conditions generated was reassuringly small, but still gave useful insights into how the system might be improved.

9. ACKNOWLEDGMENTS

We are grateful to Dr. Jay Flanz and Nghia Van Ho at Massachusetts General Hospital for their helpful explanations of the BPTC system. This research was supported by the Northrop Grumman Cybersecurity Research Consortium under the Secure and Dependable Systems by Design project, and by the National Science Foundation under grants 0541183 (Deep and Scalable Analysis of Software) and 0707612 (CRI: CRD – Development of Alloy Tools, Technology and Materials).

10. REFERENCES

[1] M. Barnett, R. DeLine, M. Fähndrich, B. J. 0002, K. R. M. Leino, W. Schulte, and H. Venter. The spec# programming system: Challenges and directions. In *VSTTE*, pages 144–152, 2005.

[2] Common Criteria Portal. Common Criteria Documents, August 2010. `http://www.commoncriteriaportal.org/thecc.html`.

[3] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *ICSE*, pages 439–448, 2000.

[4] M. B. Dwyer and S. G. Elbaum. Unifying verification and validation techniques: relating behavior and properties through partial evidence. In *FoSER Workshop, co-located with FSE*, pages 93–98, 2010.

[5] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI*, pages 234–245, 2002.

[6] P. J. Graydon, J. C. Knight, and E. A. Strunk. Assurance based development of critical systems. In *DSN*, pages 347–357, 2007.

[7] W. Griswold. Coping with crosscutting software changes using information transparency. *Metalevel Architectures and Separation of Crosscutting Concerns*, pages 250–265, 2001.

[8] D. Jackson. *Software Abstractions: Logic, language, and analysis*. MIT Press, 2006.

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

[10] D. Jackson and E. Kang. Property-part diagrams: A dependence notation for software systems. In *ICSE '09 Workshop: A Tribute to Michael Jackson*, 2009.

[11] D. Jackson, M. Thomas, and L. Millett. *Software for dependable systems: sufficient evidence?* National Academies Press, 2007.

[12] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *ISSTA*, pages 14–25, 2000.

[13] M. Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems.* Addison-Wesley, 2000.

[14] E. Kang. A framework for dependability analysis of software systems with trusted bases. Master's thesis, Massachusetts Institute of Technology, 2010.

[15] E. Kang and D. Jackson. Dependability arguments with trusted bases. In *RE*, pages 262–271, 2010.

[16] T. Kelly and R. Weaver. The Goal Structuring Notation–A Safety Argument Notation. In *Workshop on Assurance Cases, co-located with DSN*, 2004.

[17] R. R. Lutz and A. Patterson-Hine. Using fault modeling in safety cases. In *ISSRE*, pages 271–276, 2008.

[18] T. S. E. Maibaum and A. Wassyng. A product-focused approach to software certification. *IEEE Computer*, 41(2):91–93, 2008.

[19] M. Messier and J. Viega. Safe c string library, January 2005. `http://www.zork.org/safestr`.

[20] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *FSE*, pages 18–28, 1995.

[21] D. Popescu, J. Garcia, and N. Medvidovic. Enabling more precise dependency analysis in event-based systems. In *ICPC*, pages 305–306. IEEE Computer Society, 2009.

[22] A. Rae, D. Jackson, P. Ramanan, J. Flanz, and D. Leyman. Critical feature analysis of a radiotherapy machine. In *SAFECOMP*, pages 221–234, 2003.

[23] D. Smith and K. Simpson. *Safety Critical Systems Handbook: A Straightforward Guide to Functional Safety, IEC 61508 and Related Standards.* Butterworth-Heinemann, 2010.

[24] G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.*, 15(4):410–457, 2006.

[25] TIBCO (acquired Talarian in 2002). SmartSockets, August 2010. `http://www.tibco.com/products/soa/messaging/smartsockets`.

[26] W. Visser, K. Havelund, G. P. Brat, and S. Park. Model checking programs. In *ASE*, pages 3–12, 2000.