Trace Queries for Safety Requirements in High Assurance Systems

[1]Jane Cleland-Huang, [2]Mats Heimdahl, [3]Jane Huffman Hayes, [4]Robyn Lutz, [5]Patrick Maeder

[1] DePaul University, Chicago, IL 60422, USA
jhuang@cs.depaul.edu
[2] University of Minneapolis, Minneapolis, MN, USA
heimd002@umn.edu
[3] Kentucky State University, Lexington, KY, USA
hayes@cs.uky.edu
[4] Iowa State University, Ames, IA, USA, and Jet Propulsion Laboratory/Caltech
rlutz@iastate.edu
[5] Johannes Kepler University, Linz, Austria
patrick.maeder@jku.at

Abstract. [**Context and motivation**] Safety critical software systems pervade almost every facet of our lives. We rely on them for safe air and automative travel, healthcare diagnosis and treatment, power generation and distribution, factory robotics, and advanced assistance systems for special-needs consumers. [**Question/Problem**] Delivering demonstrably safe systems is difficult, so certification and regulatory agencies routinely require full life-cycle traceability to assist in evaluating them. In practice, however, the traceability links provided by software producers are often incomplete, inaccurate, and ineffective for demonstrating software safety. Also, there has been insufficient integration of formal method artifacts into such traceability. [**Principal ideas/results**] To address these weaknesses we propose a family of reusable traceability queries that serve as a blueprint for traceability in safety critical systems. In particular we present queries that consider formal artifacts, designed to help demonstrate that: 1) identified hazards are addressed in the safety-related requirements, and 2) the safety-related requirements are realized in the implemented system. We model these traceability queries using the Visual Trace Modeling Language, which has been shown to be more intuitive than the defacto SQL standard. [**Contribution**] Practitioners building safety critical systems can use these trace queries to make their traceability efforts more complete, accurate and effective. This, in turn, can assist in building safer software systems and in demonstrating their adequate handling of hazards.

Keywords: safety critical software, fault trees, traceability, visual trace queries, formal methods

1 Introduction

Requirements traceability, defined as the "ability to follow the life of a requirement in both a backward and forward direction" [6] is a critical element of any rigorous software development process. For example, the U.S. Food and Drug Administration (FDA) states that traceability analysis must be used to verify that the software design implements the specified software requirements, that all aspects of the design are traceable to software requirements, and that all code is linked to established specifications and test procedures [5]. Similarly, the Federal Aviation Administration (FAA) has established DO-178B [4] as the accepted means of certifying all new aviation software, and this standard specifies that at each stage of development "software developers must be able to demonstrate traceability of designs against requirements." Software Process Improvement standards that are being adopted by many organizations, such as CMMI, require similar traceability practices.

Traceability is broadly recognized as an important factor in building high-assurance software systems. Much of this software is safety critical, meaning that there could be devastating harm if the software fails to operate correctly. Safety-critical software systems permeate our society and are entrusted with the lives of everyday people on a daily basis. For example, a commuter on a train depends on the switching software, an airline passenger depends on the air traffic control software, and a patient in a hospital depends on the e-pharmacy software.

However, there is almost universal failure across both industry and government projects to implement successful traceability, even in safety-critical systems that require it. This has been found to be due in large part to the difficulty of constructing useful traceability queries using existing tools [18]. Traceability links may be generated at a high level, may be too generic, may be incomplete, may be inaccurate [21], and/or may not be deemed appropriate as evidence of software safety. Changes to artifacts, and hence to their traceability, often require an inordinate amount of traceability effort on the part of analysts attempting to obtain certification of even a small change to an already certified system.

The failure of traceability is of special concern in safety-critical systems where the tracking of hazards to their resolutions is mandated by certification authorities. In such systems, the traceability from hazards to software safety requirements to implemented and verified design solutions forms an essential piece of the evidence chain used to show that the resulting system is safe [1, 11, 17]. The full potential of traceability as a value-enhancing activity has not yet been realized in safety-critical systems.

To address these shortcomings, we consider the work of two stakeholder types as a safety-critical system is built, certified or modified: *developer* and *software safety engineer*. The developer prepares traditional development artifacts such as system requirements, software requirements, design (perhaps as UML diagrams), code, and test cases. Traceability matrices are generated for these artifacts (such as from system to software requirements, from code to test cases, etc.). The software safety engineer focuses on how software can contribute to a systems safety or can compromise it by putting the system into an unsafe state, and is interested in tracing the relationship between fault tree analysis results and software requirements and verification artifacts. These safety-related items also require associated traceability support.

To focus on the traceability needs of these stakeholders, this paper extends our prior work. It identifies and describes a set of twelve safety-related traceability goals that address essential traceability questions needed to demonstrate that a software intensive system meets its safety requirements. These queries cover basic life-cycle activities such as tracing from requirements to test cases, as well as more complex activities such as integrating hazard analysis and formal models and their results into the traceability environment. The trace queries are presented using the Visual Trace Modeling Language (VTML), which has been demonstrated in our prior work to be more intuitive for users to understand than the defacto standard of SQL [18]. The traceability queries are designed to deliver value-enhanced traceability in support of the producers of safety-critical software systems.

In other areas of software engineering and requirements engineering, reusable solutions, often in the form of patterns, are used to increase productivity and improve quality by capturing and applying domain knowledge to repeated problems. Traceability is no exception. Certain questions must be answered about a software system in order to achieve certification, such as “have all hazards been addressed in the requirements?” The software traceability techniques presented here help answer these questions. Like design patterns, the traceability queries are constructed to be reusable both as the system evolves and, more generally, across different systems. If modeled in advance, the traceability queries provide strategic guidance to software developers as they plan their traceability infrastructure and associated process. Reusing proven and familiar traceability queries can ease the effort of the initial certification process and provide the necessary infrastructure for supporting change, as well as helping to demonstrate safety following that change.

The remainder of paper is laid out as follows. Section 2 discusses the challenges of delivering effective traceability in a safety critical project, and introduces the concept of the Traceability Information Model (TIM). Section 3 introduces a pacemaker example, which is used to illustrate our approach. Section 4 briefly describes the VTML. Section 5 introduces and models the safety-related traceability queries, and illustrates their usefulness for the pacemaker example. Section 6 describes related work, and finally, section 7 summarizes our contribution and discusses future work.

2 Traceability in a Safety Critical Environment

Traceability decisions in a project should be documented in and driven by a traceability information model (TIM) or traceability meta-model, as depicted in Figure 1 [2, 19]. A TIM is often represented as a UML class diagram and is composed of two basic types of entities: traceable artifact types represented as classes, and the permitted trace types between the artifact types represented as associations. Traceable artifact types serve as the abstractions supporting the traceability perspective of a project.

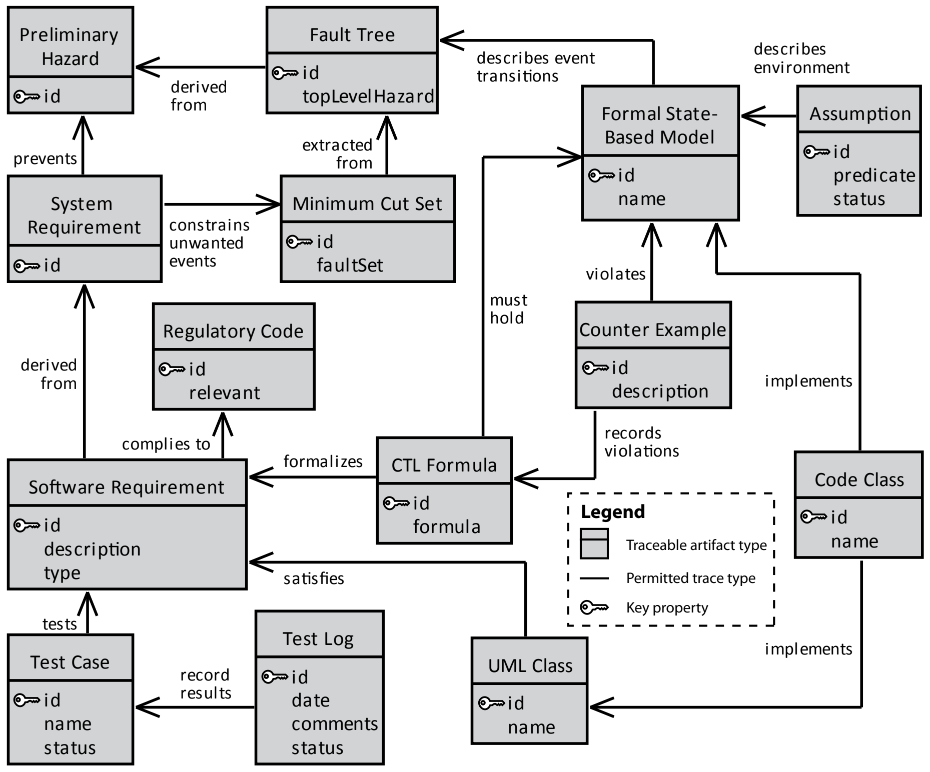


Fig. 1. A Traceability Information Model for a Safety Critical System

Figure 1 depicts the core traceable components of a safety critical system. The typical software development artifacts are seen along the left side of the diagram: system requirements are allocated to software requirements which are allocated to design elements documented as UML classes which are implemented by code. Test cases are used to test the software requirements with results being logged. Meanwhile, the safety critical nature of the software system requires additional artifacts which must also be traced, shown mainly on the right hand side of the diagram. The Preliminary Hazard artifact documents hazards that could lead to system failure. Such hazards are examined in more detail in a fault tree which looks at events that could lead to the hazards. The possible states and transitions for a system are documented in a formal state-based model. Certain assumptions about the environment are also captured. Formal analysis of the system may detect counter examples that show that a state can be entered which violates safety properties, formalized in this TIM using Computation Tree Logic (CTL). System Requirements are specified to prevent hazards from occuring by preventing the unwanted events documented in the Minimum Cut Sets. The Software Requirements may also have to comply with Regulatory Codes. Note that because this paper does not address the safety case, we have chosen not to include it in this TIM. Similarly, since we focus on product requirements, we have not shown process requirements in this TIM.

Each traceable artifact type may also possess one or more properties, which are used later in the process to specify traceability queries. For example, the "Software Requirement" artifact type includes 'id', 'description,' and 'type.' Property values can be included in trace query results, while properties or multiplicities can be used to define constraints that filter out unwanted artifacts. Filters can also be created based on trace types associated with each of the traceability paths.

Investing the effort to define a TIM is worthwhile because the TIM makes it simpler to generate and execute traceability queries. Furthermore, the TIM can be mapped to physical artifacts, and therefore a TIM and its associated trace queries can be reused across different products simply by re-establishing mappings in the new project [18]. In this paper, we present a basic TIM and define a set of reusable trace queries that are specific to the safety-critical domain.

3 Illustrative Example

We introduce a simplified pacemaker to illustrate the traceability infrastructure and to contextualize the proposed trace queries. A pacemaker [3] is an embedded medical device that monitors the heartbeat (HB) and regulates the heart when it is not beating at a normal rate. A pacemaker is safety critical because certain failures can harm the patient's health or contribute to loss of life [3, 13].

3.1 Fault Tree

One of the initial tasks in building a safety-critical software system is a preliminary hazard analysis (PHA) [12] to identify a set of potential high-level hazards, representing undesirable states of the system. System-level hazard analysis is used to help decide which hazards can be avoided (e.g., by changing the operational environment) and which hazards must be handled by the system. The PHA informs both the system safety requirements and the derived software safety requirements that constrain the design of the system. Each of the hazards in the PHA is typically explored by constructing an associated fault tree (FT) [23, 24]. A fault tree refines an initial hazard into a series of lower level intermediate or basic events, which, if they occur, would contribute toward the occurrence of the hazard. The FT uses boolean logic to depict the causal events leading to the root node. Figure 2 shows an excerpt from a fault tree constructed to investigate the ways in which the pacemaker could fail to provide treatment to the patient when needed [15].

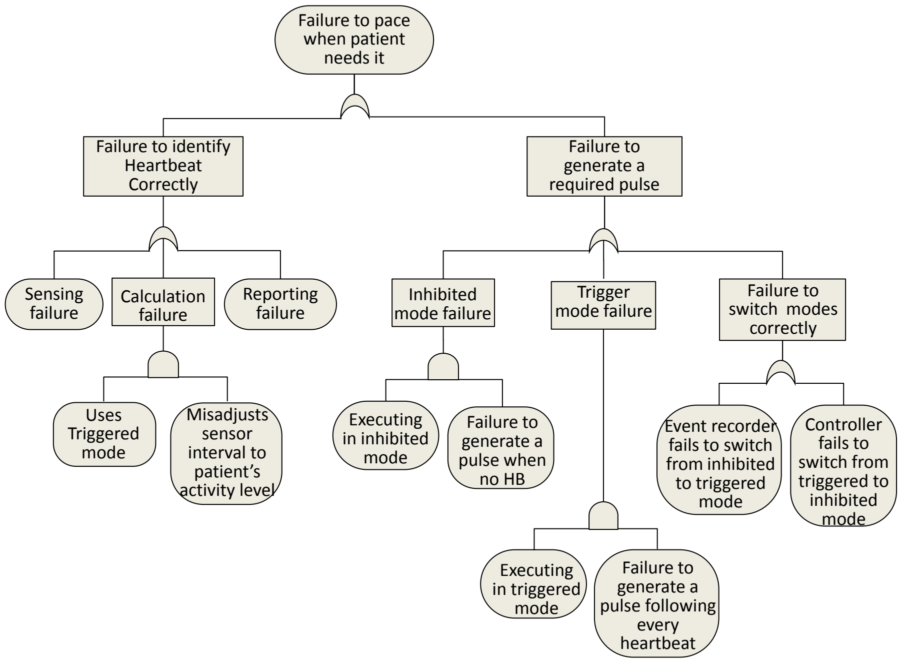

Failure to pace when patient needs it
Failure to identify Heartbeat Correctly
Failure to generate a required pulse
Sensing failure
Calculation failure
Reporting failure
Inhibited mode failure
Trigger mode failure
Failure to switch modes correctly
Uses Triggered mode
Misadjusts sensor interval to patient's activity level
Executing in inhibited mode
Failure to generate a pulse when no HB
Event recorder fails to switch from inhibited to triggered mode
Controller fails to switch from triggered to inhibited mode
Executing in triggered mode
Failure to generate a pulse following every heartbeat

Fig. 2. A Fault Tree

As depicted, the hazard under analysis is *Failure to pace when patient needs it*. Two identified intermediate faults are *Failure to identify heartbeat correctly* and *Failure to generate a required pulse*. The first of these has three contributing faults, namely sensing, calculation, and reporting failures, any one of which can cause the pacemaker to fail to pace correctly. The second intermediate fault has sub-faults related to inhibited mode failures, trigger mode failures, and transitioning from one mode to another. For purposes of this example, we are particularly interested in the inhibited mode failure which can occur when the pacemaker is in inhibited mode and there is a failure to generate a pulse when no heartbeat is detected. We are also interested in the calculation failure that occurs when triggered mode is used and the pacemaker fails to adjust the sensor interval to the patient's activity level.

A cut set in a fault tree is defined as a set of basic events (leaf nodes) whose simultaneous occurrence would cause the top event in the fault tree to occur [12]. A cut set is said to be minimum if it cannot be reduced without losing its status as a cut set. An example of a minimum cut set for the pacemaker is "failing to generate a pulse when no heartbeat is detected" while "in inhibited mode." If both of these leaf nodes occur at the same time, the pacemaker will fail to pace when needed, a hazard to the patient. Almost every fault-tree modeling tool has the capability to return the set of minimum cut sets that can be used to identify common cause failures across multiple fault trees, i.e., events that

Table 1. A Subset of Requirements for the PaceMaker System

REQ101	**Inhibited Mode**: While in inhibited mode, if no heart beat is detected by the pacemaker's sensor during a programmable sensing interval, the pacemaker shall generate a pulse.
REQ102	**Triggered Mode**: While in triggered mode, the pacemaker shall regulate the heartbeat by generating a pulse following every heartbeat.
REQ103	**Track Heartbeat Rate**: While in inhibited mode, the EventRecorder shall track the heartbeat rate.
REQ104	**Transition to Triggered Mode**: While in inhibited mode, if the heartbeat rate exceeds a threshold, the EventRecorder shall command a switch to triggered mode.
REQ105	**Transition to Inhibited Mode**: While in Triggered mode, if the number of heartbeats exceeds 24 in a 6000 msec recording interval, the Controller shall command a switch to Inhibited mode.
REQ106	**Activity Sensor**: The pacemaker shall monitor the activity level of the patient.
REQ107	**Activity Response**: The pacemaker shall adjust the duration of the sensing interval to match the patient's current activity level.

occur in the minimum cut sets of multiple fault trees. In addition, some tools can return common cause events.

3.2 Safety-Related Software Requirements

The basic functionality of the pacemaker involves two different operation modes: inhibited and triggered [14]. In inhibited mode, the pacemaker generates a pulse only if the heart fails to generate its own pulse, while in triggered mode, the pacemaker generates a pulse following each heartbeat. Some pacemakers, such as the one illustrated here, also have the ability to monitor the activity level of a patient in order to adjust the sensing interval accordingly. These requirements are more formally depicted in Table 1. Note that these requirements may be found as a subset of the System or Safety Requirements from the TIM shown in Figure 1.

3.3 Safety Analysis

Once failure causes are well understood and the software requirements to address these (called *software safety requirements*) are specified and validated, developers construct the design to satisfy the requirements and produce code to implement the design. Certain properties must be satisfied by the pacemaker design and implementation in order to assure patient safety. Moreover, these properties must be shown to be satisfied in order for the company producing the pacemaker to gain approval to market and sell their devices. An example of such a safety-related property is requirement REQ101 related to pulse generation. An examination of the fault tree in Figure 2 shows that this property is the inverse of the minimum cut set containing the two leaf nodes "Fails to generate a pulse" and "Is in inhibited mode."

Many of the safety engineer's tasks thus involve assurance that traceability exists between the safety requirements and the intermediate and final products.

Some of the assurances the safety engineer is responsible for providing involve relatively straightforward queries such as “Are all initially identified hazards covered by a fault tree?” Other assurances involve more complicated traceability queries such as “Do all minimum cut sets have an associated mitigating requirement?” or “Are all common cause failures in the set of fault trees addressed by one or more design mechanisms?” In previous work we presented a set of eleven standard trace query patterns needed for the assurance of requirements for an e-health software system that did not have explicit safety requirements [18]. In this paper we extend those queries to include trace queries needed to handle the assurance of software safety requirements.

For each trace query, we describe how the query is represented using our Visual Trace Modeling Language (VTML), and discuss the results returned by an example of the traceability query for the pacemaker. Each of these queries addresses a common question that must be repeatedly posed by either a safety engineer or a developer in the safety-critical domain, for which current techniques generally require significant manual effort to answer. Representation of the queries in VTML enables the associated queries to be used and reused across the artifacts in the TIM. If a query returns bad news, the safety engineer can place this item on a watch list. New queries then can be periodically run behind the scenes. If new fault trees are identified or existing fault trees are updated in response to evolution in requirements, design, or operational experience [17], the safety engineer can perform a delta trace to determine if added or modified hazards are adequately covered.

4 Visual Trace Modeling Language (VTML)

We illustrate the trace queries in this paper using VTML. VTML assumes the presence of an underlying TIM and then represents queries as a set of filters applied to a structural subset of that model. A VTML query is composed of a connected subset of the artifacts and trace types defined in the TIM as well as a set of associated filter conditions. These filters are used to eliminate unwanted artifacts or to define the data to be returned by the trace query.

Figure 3 depicts the basic elements of a VTML query. The initial query scope specifies the subset of artifacts for which the trace is to be executed, where scope could be as small as a single artifact, or as broad as the entire set of artifacts of that type. VTML depicts this scope visually using the *start* symbol. The three compartments of the class notation are used respectively to depict the name of the class, properties used in filter conditions or to specify return results, and functions used to compose and extract aggregate data from the class. Return values are annotated with a bar chart symbol, while properties used to filter results are annotated with a filter symbol and also depict a valid filter expression. As shown in this example, filters can be applied at both the class and the trace matrix level. The example in Figure 3 can be read as follows assuming source artifacts are *use cases* and target artifacts are *test cases*: “For the selected use cases, return the description of all use cases which trace to more than two failed

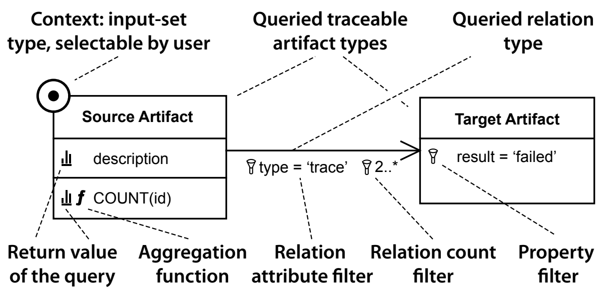


Fig. 3. Features of a visual traceability query

test cases. Aggregate the results according to some function *f*, and display the description and the aggregated value." A more complete description of VTML including its metamodel and an extensive set of queries is provided in our prior work [18].

5 Safety-Related Trace Queries

Traceability provides support for specific software engineering goals, as depicted in Table 2. These goals are derived from a number of sources including Leveson's set of basic software system safety tasks [12], our own experiences working with safety-critical systems [7,16], an analysis of several documents prepared as submissions for approval of medical devices, and a study of related literature, handbooks, and guides [9].

For each of these traceability goals, there are several different supportive traceability queries that can be used by the safety analyst. For example, if we are interested in Traceability Goals #2 (safety-related requirements covered by design) and #6 (safety-related requirements have been tested), we might focus on tracing *requirements* to *code*. Queries of interest might include (a) "return a list of all requirements and the associated classes in which they are implemented", and (b) "count the number of requirements without implemented classes." These queries reveal something about the coverage of requirements in the implementation. Similarly, (c) "return a list of all requirements without associated implemented classes" or (d) "count the number of requirements without implemented classes" both reveal information about lack of coverage. We could also execute transitive trace queries such as (e) "return a list of all requirements with classes that have failed test cases in the past week," or we could incorporate customized functions into the trace queries as (f) "return a list of requirements with classes that exhibit cyclomatic complexity values in the top 5 percentile."

As it is not feasible for us to illustrate each type of query for each of the twelve proposed trace queries, we illustrate our approach with trace queries for three of the goals that are particularly relevant to the safety-domain, and which are quite different from queries found in non-safety critical domains. All of these queries assume the underlying presence of the TIM depicted in Figure 1.

Table 2. Safety-Related Traceability Goals

1.	Demonstrate that all common cause failures in the set of fault trees are covered by requirements.
2.	Demonstrate that all safety-related requirements are satisfied in the design.
3.	Determine which regulatory codes are covered by requirements.
4.	Demonstrate that all safety-related design elements are fully realized in the code.
5.	Identify parts of the code which represent standard safety mechanisms including architectural or design mechanisms such as safety interlocks, heartbeat or fault-data redundancy, to prevent a specific hazard from occurring.
6.	Demonstrate coverage of safety-related requirements by test cases.
7.	Demonstrate that safety-related test cases have passed.
8.	Demonstrate that properties specifying safety-related requirements to be model checked have been model checked.
9.	Demonstrate that all counter-examples produced by the formal model checker for any of the safety-related requirements have been reviewed by a safety engineer.
10.	Determine the potential impact of changing a requirement on its associated downstream, safety-related TIM artifacts.
11.	Determine which requirements might be impacted by failure of a safety-related test case.
12.	Determine which formal models might be impacted by a change to an environmental assumption.

5.1 Requirement Coverage of all Common Cause Failures

In support of traceability goal # 1, it is important to show that all minimum cut sets derived from the modeled fault trees are covered by requirements. Showing that each minimum cut set is associated with one or more mitigating requirements can provide a safety engineer with the information he or she needs to assess whether the hazard is fully mitigated. We present an example of one supporting trace query in Figure 4. This query returns a list of minimum cut sets and their associated requirements for one or more fault trees. As the VTML assumes a default cardinality of *1..**, the query only returns the minimum cut sets which have related system and software requirements. A similar query in which a cardinality filter of *0* is placed on the link between *Minimum Cut Set* and *System Requirement* would list only the minimum cut sets without system level requirements coverage.

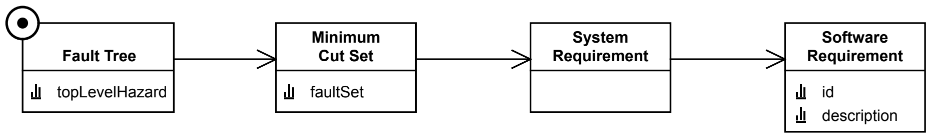


Fig. 4. Trace Query: Retrieve requirements providing coverage for minimum cut sets derived from one or more fault trees.

Applying the trace in Figure 4 to the pacemaker example produces a trace matrix that includes the entries depicted in Table 3. These traces not only demonstrate that the minimum cut sets are associated with software requirements, but provide the safety engineer with information needed to assess how well they mitigate the common cause failures.

Table 3. A Subset of Results Returned by the Minimum Cut Set Coverage Query

Fault Tree	Minimum Cut Set	System Requirement	Software Requirement
Failure to pace when patient needs it	(i)executing in inhibited mode,(ii)failure to generate a pulse when no HB	Monitor battery power to ensure pulse can be given.	Log failure event internally for diagnosis; Send wireless phone warning to health provider upon recurrence.
Failure to pace when patient needs it	(i)uses triggered model, (ii)adjusts sensor interval to patient's activity level	Activity sensors are monitored at all times for correct function.	If the respiration sensor (indicating activity level) fails, the pacemaker shall use Inhibited mode

5.2 Integrating Formal Method Results

There is an increasing trend in safety-critical software development toward more formally verifying the correctness of the design through model checking [14]. However in current practice, the model checking results are often disconnected from other software artifacts and are therefore often not used in the traceability scheme. In this section we propose a trace queriy for integrating model checking results into the TIM in support of Trace Goal #8. The query depicted in Figure 5 utilizes the formal model components of the TIM. First, it identifies any counter examples produced by the model checker. If any are identified, it returns a list of the associated CTL formulas and related requirements.

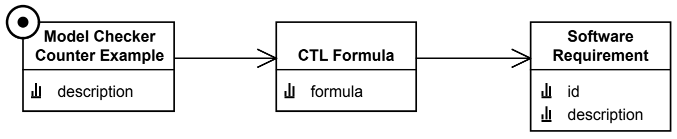


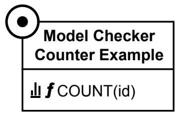
Fig. 5. List all CTL formulas and related requirements for any counter examples produced by the model checker

To illustrate this query, consider the pacemaker requirement REQ101 which states that "While in inhibited mode, if no heart beat is detected by the pacemaker's sensor during a programmable sensing interval, the pacemaker shall generate a pulse." An associated CTL could be defined as follows [14]:

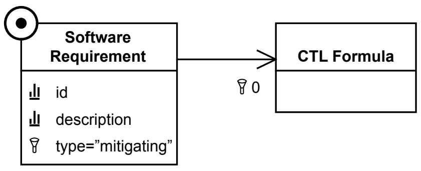
$$AG((sensed = 0 \ \wedge \ timerSenseTimeUp = 1 \ \wedge \ inhibitedMode = 1)) \\ \implies EF(pulseGen = 1 \ \wedge \ inhibitedMode = 1))$$

This and similar CTL properties are checked by the model, and results are stored in a model checking repository. Assuming no counterexamples are produced, the query in Figure 5 returns an empty list, adding some degree of confidence that given the as-modeled behavior of the system, this requirement is always satisfied.

Figure 6 depicts two additional kinds of supporting trace queries for counting artifacts and for identifying missing elements. The first shows how a trace query can be used to return a simple count of counter examples produced by the most recent model checking run, while the second one returns a list of mitigating requirements without associated CTL formulas. Both of these trace queries and their results can be used by a safety engineer to help manage safety requirements throughout the software development effort.



(a) Query 3a: Return a count of counter examples produced by the most recent model checking run



(b) Query 3b: Return a list of mitigating requirements without associated CTL formulas

Fig. 6. Supporting Traces for Integrating Results from the Model Checker

5.3 Assumptions

In our final example we present a trace query that supports Goal #12. Each formal model typically has a set of assumptions associated with it. These assumptions are often in the form of predicates such as "A patient's heartbeat is always (can be assumed to be) in the range x to y." or "the sensor that checks the patient's respiration rate never (can be assumed to never) fails." Sometimes during use of the system, or due to changes in the environment, these assumptions are found to be, or become, incorrect. The properties verified on that model were based on those assumptions, so we can no longer be confident in safety arguments based on the model. In the trace query depicted in Figure 7, we therefore retrieve a list of all CTL properties and associated requirements that are impacted by a change in one or more assumptions.

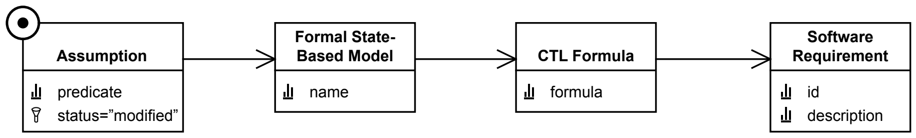


Fig. 7. Trace Query:List all requirements impacted by a change in an environmental assumption and the formal models that must be re-checked

5.4 Prototype

One of the major benefits of VTML is that trace queries are defined over the TIM, and do not reference project-specific data structures. However, the queries must be transformed into a query format that can be applied to the physical data sources. All of the trace queries described in this paper are fully executable in our prototype tool. Our prototype transforms the features of a visual traceability query step by step into an executable SQL query. It first uses an XSLT script that translates queries into XMI format, and then transforms them into executable SQL statements [18]. Defining and writing trace queries using VTML applied over a standard TIM, makes the queries fully portable across projects. It means that an organization adopting our appproach could create both a reusable TIM and a reusable set of safety-related trace queries which address all of the traceability goals defined in Table 2. This portability is achieved by mapping the conceptual elements of the TIM, including the artifact types and their properties, to physical fields in the underlying database.

6 Related Work

Most discussion of traceability in the development of safety-critical systems is in the form of standards and guidebooks that mandate the tracking of hazards and their mitigations through the software life cycle but do not describe query techniques to help achieve this. However, safety cases [11], dependability cases [1], and assurance cases all use traceability to construct structured arguments to justify goals by tracing and managing the links from evidence to those goals. Recommended practice is to maintain the case while constructing the system so that every step of development preserves the established chain of evidence. Although there is a large body of work in the more general area of traceability, to the best of our knowledge, there is little or no research that investigates techniques for using traceability to support a broad spectrum of safety-related queries in the way described in this paper. Extending the work described here to support assemblage and maintenance of safety case evidence is a natural and planned extension.

Peraldi-Frati and Albinet proposed a model for traceability in safety-critical systems [20]. Their work focused on requirements, design, and test cases, and showed how to establish *satisfies* relationships from design to requirements, and *verifies* relationships between test cases and requirements. Their proposed infrastructure incorporates formal models that demonstrate the satisfaction of a specific requirement. Katta and Stalhane define a conceptual model of traceability for safety systems [10]. Their approach creates a traceability graph (similar to a TIM) depicting a wide variety of artifacts and their associated traceability links. For example, they include hazards, system level requirements, software requirements, architectural components, and common cause failures. However, neither of these approaches incorporates results from fault tree analysis nor integrates formal methods into the traceability infrastructure. Furthermore, in general, any publications we found on tracing safety-critical requirements focus upon describing the actual artifacts to be traced, and fail to highlight the tracing benefits achieved through a useful and effective set of traceability queries.

Hill and Tilley propose a traceability approach for supporting the assurance and recertification of safety-critical legacy systems [8]. However, they primarily describe traces between requirements, process improvement standards, and a risk taxonomy and do not discuss any specific types of software artifacts beyond requirements. Finally, other researchers such as Sanchez et al. have explored the role of traceability in safety-critical, model-driven environments [22]. Their approach is designed to demonstrate that hazards translate into requirements, and that architectural decisions designed to satisfy those requirements are successfully transformed into the final code.

7 Conclusions

The traceability goals and queries described in this paper support a number of critical safety engineering tasks. First, they can be used during the development process to ensure that safety is being built into the system, and second, they can be used to generate traceability matrices needed by certification and approval bodies such as the FDA. Combining the various types of coverage queries produces relatively sophisticated and clearly useful trace matrices. It also identifies problem areas such as safety-related requirements without passed test cases, or safety-related requirements potentially impacted by changed values of environmental variables which provide significant support towards building a demonstrably safe software system.

The primary contribution of this paper is the presentation of a query-driven approach to tracing requirements in safety-critical software systems. At the start of a project, safety engineers and developers can strategically plan the TIM, map it to specific database tables or other data structures, and carefully define the safety-related trace queries that are to be accessible throughout the project. This kind of approach enables engineers to build traceability into the software development life-cycle, so that traceability links can be used not only for documentation purposes during the certification process, but for actually improving developers' understanding of safety-related issues throughout the software development life-cycle.

8 Acknowledgments

This work was supported by NSF grants CCF-0916275 with funds from the American Recovery and Reinvestment Act of 2009, CCF-1143830, CCF-1143734, CCF-0810924 and CCF-0811140. This research is also funded in part by the Austrian Science Fund (FWF): M1268-N23.

References

1. D.Jackson, M. Thomas, and L.I.Millet. In *Software for Dependable Systems: Sufficient Evidence?, National Research Council*, 2007.
2. R. Dömges and K. Pohl. Adapting Tracability Environments to Project-Specific Needs. *Communications of the ACM*, 41(12):54–62, dec 1998. ISSN 0001-0782.
3. K. A. Ellenbogen and M. A. Wood. *Cardiac Pacing and ICDs.* Blackwell Publishing, 2005.
4. Federal Aviation Authority (FAA). *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, faa's advisory circular ac20-115b edition.
5. Food and Drug Administration. *Guidance for the Content of Premarket Submissions for Software Contained in Medical Devices*, 2005.
6. O. Gotel and C. Finkelstein. An analysis of the requirements traceability problem. In *Requirements Engineering, 1994., Proceedings of the First International Conference on*, pages 94 –101, apr 1994.
7. M. P. E. Heimdahl. Safety and software intensive systems: Challenges old and new. In *FOSE*, pages 137–152, 2007.
8. J. Hill and S. Tilley. Creating safety requirements traceability for assuring and recertifying legacy safety-critical systems. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*, pages 297 –302, 27 2010-oct. 1 2010.
9. Joint Software System Safety Committee. *Software System Safety Handbook Technical and Manegerial Team Approach*, 1999 edition.
10. V. Katta and T. Stalhane. A conceptual model of traceability for safety systems. In *CSDM - Poster Presentation*, 2010.
11. T. P. Kelly and J. A. McDermid. A systematic approach to safety case maintenance. In *SAFECOMP*, pages 13–26, 1999.
12. N. G. Leveson. *Safeware, System Safety and Computers.* Addison Wesley, 1995.
13. B. Littlewood and L. Strigini. Validation of ultrahigh dependability for software-based systems. *Commun. ACM*, 36(11):69–80, 1993.
14. J. Liu, S. Basu, and R. Lutz. Generating variation point obligations for compositional model checking of software product lines. In *Journal of Automated Software Engineering*, pages 39–76, Vol. 18 (1),2011.
15. J. Liu, J. Dehlinger, H. Sun, and R. R. Lutz. State-based modeling to support the evolution and maintenance of safety-critical software product lines. In *ECBS*, pages 596–608, 2007.
16. R. R. Lutz. Software engineering for safety: a roadmap. In *ICSE - Future of SE Track*, pages 213–226, 2000.
17. R. R. Lutz and I. C. Mikulski. Requirements discovery during the testing of safety-critical software. In *ICSE*, pages 578–585, 2003.
18. P. Mäder and J. Cleland-Huang. A visual traceability modeling language. In *MoDELS (1)*, pages 226–240, 2010.

19. P. Mäder, O. Gotel, and I. Philippow. Getting Back to Basics: Promoting the Use of a Traceability Information Model in Practice. In *5th Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE2009). In conjunction with ICSE09*, Vancouver, Canada, May 2009.
20. M.-A. Peraldi-Frati and A. Albinet. Requirement traceability in safety critical systems. In *Proceedings of the 1st Workshop on Critical Automotive applications: Robustness &#38; Safety*, CARS '10, pages 11–14, New York, NY, USA, 2010. ACM.
21. B. Ramesh and M. Jarke. Toward reference models for requirements traceability. *IEEE Trans. Softw. Eng.*, 27:58–93, January 2001.
22. P. Sánchez, D. Alonso, F. Rosique, B. Álvarez, and J. A. Pastor. Introducing safety requirements traceability support in model-driven development of robotic applications. *IEEE Trans. Computers*, 60(8):1059–1071, 2011.
23. N. R. Storey. *Safety Critical Computer Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.
24. K. J. Sullivan, J. B. Dugan, and D. Coppit. The galileo fault tree analysis tool. In *FTCS*, pages 232–235, 1999.