Constructing Bayesian-network Models of Software Testing and Maintenance Uncertainties

To appear in ICSM'97 International Conference on Software Maintenance

Hadar Ziv
Debra J. Richardson
Department of Information and Computer Science
University of California, Irvine
Irvine, California 92697-3425

Phone +1.714-824-4047
FAX +1.714-824-4056

ziv@ics.uci.edu
djr@ics.uci.edu

UCI-ICS Technical Report 97-23

Abstract

The lifetime of many software systems is surprisingly long, often far exceeding initial plans and expectations. During development and maintenance of long-lived software, requirements are analyzed and specified, designs and code modules are developed, testing is planned, and code is tested many times. Consequently, developers and managers frequently lose or gain confidence in software artifacts, especially when existing uncertainties are relieved or when new uncertainties are encountered. Fluctuations in developers' confidences may in turn affect process actions or decisions, for instance determining the impact of change, the need for regression testing, or when to stop testing. In this paper, we present an approach that allows for developers' confidences or "beliefs" regarding software components to be modeled and updated directly. This approach is part of an overall strategy that calls for explicit modeling of software engineering uncertainties using an established technique for uncertainty modeling called Bayesian belief networks. Here, we present several types of software uncertainty and how they may be modeled directly. We also introduce Bayesian belief networks and how they may be used to either confirm, evaluate or predict software uncertainties. We discuss our experiences with constructing Bayesian-network models of uncertainty for an existing system developed at Beckman Instruments. Once constructed, these models may be used by developers and managers in future software understanding, evolution, and maintenance activities. We also list factors affecting levels of confidence in system artifacts, identified in interviews with Beckman developers. Finally, we describe our design and implementation of a Java program that allows software systems and associated beliefs to be modeled explicitly.

Keywords: Uncertainty modeling, Bayesian belief networks, Software requirements, Software testing, Software maintenance

1 Introduction

Development of complex software is fraught with uncertainties. These uncertainties in turn lead to various software defects, symptoms of the "software crisis," and occasionally to harmful failures such as reported for the Therac-25 medical radiation device [LT93] and the Ariane-5 missile launch attempt [LB96]. Significant efforts in software engineering research are aimed at relieving and minimizing uncertainties, though removing them entirely is generally impossible. Despite such research efforts, many software development activities, including requirements specification, coding, testing and maintenance, are typically carried out in an ad hoc fashion, especially when restricted by development budgets, available resources, and time-to-market constraints. In real-life software projects, testing is often done after coding is complete, does not correspond to original requirements, and does not provide for adequate coverage nor proper partitioning of the input domain.

Nevertheless, software requirements are specified, designs and source code modules are developed, test plans are prepared, test suites are created, code is tested and later retested frequently during system evolution. Because of uncertainties, developers are seldom entirely confident in their software artifacts. Instead, they typically retain, at least intuitively, some degree of confidence in those artifacts. Developers' confidence levels tend to fluctuate during development and maintenance, often when existing uncertainties are relieved or new uncertainties are introduced. Key software process concerns, such as "Are requirements specified accurately and completely?" and "When to stop testing?", are decided, at least partially, based on developers' confidences in requirements and test artifacts, respectively.

In this paper, we contend that software managers and developers alike would benefit immensely from explicit modeling of their confidences in software artifacts. Since confidences decrease and increase due to uncertainties, we propose that developers' confidences be captured using techniques for uncertainty modeling. This proposal follows an earlier observation [ZR97a] that software uncertainties exist, are ubiquitous, are relevant to development steps and process decisions, yet surprisingly are seldom managed and modeled. Our approach calls for software uncertainties to be captured as probability values, using an uncertainty-modeling technique known as Bayesian belief networks (originally described by Pearl [Pea88]). This approach requires only that program source code, along with related specifications, designs and test information, are known and available, and that initial confidence levels can be determined. Confidence levels are obtained either objectively, for instance based on statistical or historical data, or subjectively, for instance by interviewing domain experts. Explicit modeling of individual software artifacts, relations, and associated uncertainties, is but one example of larger strategy that calls for software uncertainties to be modeled using Bayesian belief networks. Littlewood and Strigini, for instance, adopt a Bayesian approach to modeling confidences in program reliability and dependability [LS93]. In its most general form, therefore, this strategy is applicable to many software situations where uncertainty is present and its modeling deemed beneficial.

To facilitate future construction of Bayesian models, we have implemented Bayesian-network capabilities for software systems. We describe our design and implementation of a java program that allows software systems to be defined as networks, called Software Belief Networks or simply SBNs, of interrelated artifacts with associated belief values. After initial construction, an SBN becomes subject to Bayesian updating, as defined by Pearl [Pea88]. To afford Bayesian updating, we employ an existing system, also implemented in Java and available on the WWW, called JavaBayes [Coz96]. For legacy software, the corresponding SBN is maintained and updated simultaneously with the system itself.

In this paper, we describe the construction of Bayesian models of software uncertainties. These models represent developers' confidences in software artifacts for a system currently under development at Beckman Instruments in Fullerton, California. This system, called CEquencer, controls and communicates with hardware devices used by biologists, chemists, and other scientists to separate laboratory specimens into molecular constituents to help determine their DNA sequences. It is important to note that CEquencer is developed largely by reusing and augmenting previous designs and implementation of similar systems at Beckman.

This paper is organized as follows: First, relevant aspects of uncertainty modeling in software engineering are presented, followed by detailed discussion of software requirements and testing uncertainties, including examples of modeling those uncertainties. We then introduce Bayesian belief networks and how they can be used to either confirm, evaluate, or predict software uncertainties. Specific examples of Bayesian-network construction are provided for Beckman's CEquencer system. We also introduce the notion of confidence factors, which may influence developers' confidences in software artifacts; seven such factors were identified with respect to CEquencer requirements. Lastly, we describe our design and implementation of a Java program that supports the construction of Bayesian-network models of software systems.

2 Software Engineering Uncertainties

Here, we present three aspects of software uncertainty deemed most pertinent to this paper, namely categories of applying uncertainty modeling, requirements analysis uncertainty, and testing uncertainty. The reader is referred to [ZR97a] for many additional details.

2.1 Uncertainty Modeling in Software Engineering

In [ZR97a], we claim that uncertainty abounds in software development, stated succinctly as the Maxim of Uncertainty in Software Engineering (MUSE): *"Uncertainty is inherent and inevitable in software development processes and products."* We also propose, as a corollary to MUSE, that software uncertainties should be modeled and managed explicitly, preferably using an established uncertainty-modeling technique such as Bayesian belief networks. This is followed by identifying three broad categories of applying uncertainty modeling to software engineering tasks, as follows:

1. Confirmation: Certain characteristics or behaviors of software systems would seem reasonable to most developers and are therefore expected to hold. For instance, confidence in the correct behavior of a program is expected to increase if no defects are detected by test case execution and decrease otherwise. Uncertainty modeling can be used to confirm such expectations (as in [ZR97a]). For purposes of confirmation, Bayesian-network models are relatively easy to construct, yet correspondingly are of limited value to developers.

2. Evaluation: Of more interest is evaluation of whether desirable software qualities or properties are indeed present. One may have, for instance, created regression test suites based on one or more test adequacy criteria, such as described in [CPRZ89]. Uncertainty still exists, however, regarding true defect-detection abilities of those regression test suites (cf. [FW93]). This uncertainty may be modeled using uncertainty-modeling techniques. The resulting model, be it Bayesian or otherwise, may be used to evaluate the test suite's defect detection ability.

3. Prediction: Predicting certain qualities or properties of planned development activities or artifacts is most difficult but also most beneficial to developers. Consider, for example, the following change management scenario: Given new system requirements and corresponding new designs, one ultimately wishes to predict the quality of resulting code. Project managers,

for example, like to know in advance which code segments are more time-consuming or error-prone. This prediction task may be accomplished by means of uncertainty modeling.

2.2 Requirements Analysis Uncertainty

Successful software development is often hindered by the generally poor state of most requirements descriptions. Software requirements analysis typically includes learning about the problem and problem domain, understanding the needs of potential users, and understanding the constraints on the solution. Investigations of the software crisis indicate that poor up-front definition of requirements is one of the major causes of failed software efforts [Pre92]. This hindrance to successful software development is captured eloquently in Humphrey's *requirements uncertainty principle* [Hum95]: "For a new software system, the requirements will not be completely known until after the users have used it." In this paper we suggest that, given the inherent uncertainties of specifying requirements, software analysts will do well to record those uncertainties explicitly, rigorously and accurately.

Examples of software requirements uncertainties include, among others: Who are the real system users? What precisely are users' needs and expectations? How well are they represented in the requirements document? How well is the problem domain understood? How well is it captured in the requirements document? Additional uncertainty is then introduced in the transition from requirements specification through design to coding and system integration. Development of complex software often requires the system to be represented at multiple levels of abstraction, including requirements specifications, architectural and other design models, and source code implementations. Transitioning between different levels of abstraction, however smooth, often introduces uncertainties, including: How well does the design model correspond to the requirements analysis model? How well does the implementation correspond to the design? How many of the specified requirements are indeed met?

2.3 Software Testing Uncertainty

Software testing is fraught with uncertainty due primarily to the inherent inability to completely determine correctness by testing. Testing requires both planning and enactment, where enactment includes test selection, test execution, and test result checking. Test enactment is inherently uncertain, since only exhaustive testing in an ideal environment guarantees total confidence in the testing process and its results. This ideal testing scenario is infeasible for all but the most trivial software systems. Instead, multiple factors exist that introduce software testing uncertainties.

2.3.1 Test Planning

We identify three aspects of test planning where uncertainty is present: the artifacts under test, the test activities planned, and the plans themselves. Software systems under test include, among others, requirements specifications, produced by requirements elicitation and analysis; design representations, produced by architectural and detailed design; and source code, produced by coding and debugging. Since uncertainty permeates these processes and products, plans for their testing will inevitably carry those uncertainties forward. In section 3 we show that Bayesian modeling of testing uncertainties supports their forwarding by way of Bayesian updating.

In addition to requirements, design, and coding uncertainties, software testing is also human-intensive and thus introduces its own uncertainties. These uncertainties may in turn affect the development effort and should therefore be accounted for in the test plan. In particular, many testing activities, such as test result checking, are highly routine and repetitious and thus are likely to be error-prone if done manually, which introduces additional uncertainty. Test planning activities are carried out by humans at an early stage of development, thereby introducing uncertainties into the resulting test plan. Also, test plans are likely to reflect uncertainties that are, as described above, inherent in software artifacts and activities. During planning, probability values that correspond to testing uncertainties are typically estimated either based on prior, historical data, or by experts' subjective assessments.

2.3.2 Test Selection

Test selection is the activity of choosing a finite set of elements (e.g., requirements, functions, paths, data) to be tested out of a typically infinite number of elements. Test selection is often based on an adequacy or coverage criterion that is met by the elements selected for testing. The fact that only a finite subset of elements is selected inevitably introduces a degree of uncertainty regarding whether all defects in the system can be detected. One can therefore associate a probability value with a testing criterion that represents one's belief in its ability to detect defects. An example of assigning confidence values to path selection criteria is given below.

Substantial efforts in software testing research have been devoted to defining testing techniques and later evaluating their relative merits in detecting software defects. These efforts can be classified roughly into experimentation, simulation, and analysis, including evaluation and analysis of random testing [DN84] and partition testing [RC85, WJ91, JW89, HT90]; subsumption hierarchies for data flow testing criteria [CPRZ89, FW88a, FW88b, WWH91]; and subsequent improvements to subsumption. Particularly unsettling are Hamlet and Taylor's results that "partition testing does not inspire confidence" [HT90] and Frankl and Weyuker's results that a subsumption relations among test criteria do not necessarily guarantee superior defect detection abilities [FW93].

2.3.3 Example: Path Selection Testing Criteria

To model the uncertainties associated with test criteria, we associated confidence levels with the path selection criteria in [CPRZ85, CPRZ89]. There, the authors present a subsumption hierarchy that suggests a partial order of data flow path selection criteria regarding their ability to provide adequate coverage of a given program. Subsumption implies relative strength of test criteria, which may be recast as confidences, as follows: If criterion A subsumes criterion B, then A is viewed as superior to B with respect to defect detection. As a result, a higher level of confidence would typically be associated with A's defect detection abilities than those of B. Note that, as discussed in [CPRZ89], even if A subsumes B, uncertainty still remains whether A is in fact better than B, since demonstrating A's superior defect detection abilities would require that empirical data be collected to substantiate the graph theoretic proofs of subsumption. This is discussed further in [FW93], where it is shown that subsumption does not necessarily guarantee superior defect detection abilities.

Confidence in defect detection abilities of a given testing criterion may be quantified by means of a probabilistic belief value between 0 and 1. We use "belief" rather loosely here, but later we distinguish between "confidence", referring to a subjective, typically from a human perspective, measure, and "belief," referring to terminology used specifically in Bayesian-network modeling. A plausible assignment of probabilistic confidence values to a dozen path selection criteria from [CPRZ89], based on input from a domain expert, was given in [ZR97a]. These confidence values were then used to confirm developers' expectations for an elevator control system. Here, assigned confidence values are summarized in Table 1.

Noticeably, in this example, confidence values assigned to path selection criteria are relatively low. Low confidence values indicate that even "strong" path selection criteria do not necessarily

Path Selection Criterion	Confidence Value
All-Paths	.65
All-DU-Paths	.59
Ordered Context Coverage+	.61
Context Coverage+	.55
Reach Coverage+	.45
All-Uses	.45
All-C-Uses/Some-P-Uses	.33
All-P-Uses/Some-C-Uses	.33
All-Defs	.25
All-P-Uses	.2
All-Edges	.15
All-Nodes	.1

Table 1: Confidence Values for Data Flow Path Selection Criteria

incur high degree of confidence in their defect detection abilities. This is because, in addition to subsumption uncertainties discussed above, path selection does not take into account, for instance, data value selection. Some defects are only revealed by certain data values, but not by others. Low confidence values for path selection criteria therefore reflect the criteria's inability to guarantee superior defect detection capabilities.

2.3.4 Test Execution

Test execution involves actual execution of system code on some input data. Test execution may introduce uncertainties, as follows: the system under test may be executing on a host environment different from the target execution environment, thereby introducing uncertainty. Also, in cases where the target environment is simulated on the host environment, testing accuracy can only be as good as simulation accuracy. Furthermore, observation may affect testing accuracy with respect to timing, synchronization, and other dynamic issues. Finally, test executions may not accurately reflect the operational profiles of real users or real usage scenarios.

2.3.5 Test Result Checking

Test result checking is likely to be error-prone, inexact, and uncertain. Test result checking is afforded by means of a test oracle, that is used for validating results against stated specifications. Test oracles can be classified into five categories [RAO92], offering different degrees of confidence, ranging from human oracles (lower confidence) to specification-based oracles (higher confidence). Even specification-based oracles, though inspiring high confidence, are susceptible to uncertainties due to discrepancies between the specification and customer's actual needs and expectations.

3 Bayesian Models of Uncertainty

3.1 Introduction to Bayesian Belief Networks

Bayesian belief networks [Pea88, Nea90] have been used in artificial intelligence research as framework for modeling and reasoning with uncertainty. Several applications of Bayesian (or closely related) techniques are currently in use, including interpretation of live telemetry data, power generation monitoring, real-time weapons scheduling, medical diagnosis systems, and many other diagnostics applications (see [iAA96, HMW95]). Of particular relevance are successful applications of Bayesian networks to large text and hypertext search databases in the domain of information retrieval [Fri88, Cro93] and to validation of ultrahigh dependability for safety-critical systems [LS93].

Generally, a Bayesian belief network offers a graphical presentation of causal, probabilistic relationships among variables. The graphical depiction is a Directed Acyclic Graph (DAG), where graph nodes represent variables whose values come from discrete or "enumerated" domains. In the following, we use "nodes" when discussing structural aspects of Bayesian networks and "variables" when discussing probabilities.

Directed edges between nodes represent causal influence. Each edge has an associated matrix of probabilities to indicate beliefs in how each value of the cause (i.e., parent) variable affects the probability of each value of the effect (i.e., child) variable. The structure and probability values for a Bayesian network are determined by the application domain and guided by consultation with experts. Edge-matrix probabilities can either be estimated by experts or compiled from statistical studies. An important assumption of Bayesian networks is variable independence: a variable depends (in the probabilistic sense) only on its parents.

Bayesian updating occurs whenever new evidence arrives. Here, we follow Pearl's original updating algorithm [Pea88], based on a message passing model, where probability vectors are sent as messages between network nodes. Bayesian updating proceeds by repeatedly sending messages, both "up" the network from a child node to its parent and "down" the network from a parent node to its child, until all nodes are visited and their belief values, if needed, revised. As discussed in section 5, this message-passing updating scheme is conducive to distributed implementation.

A Bayesian belief network is defined formally as a triplet (N, E, P), where N is a set of nodes, $E \subseteq N \times N$ a set of edges, and P a set of probabilities. Each node in N is labeled by a random variable v_i, where $1 \leq i \leq |N|$. Each variable v_i takes on a value from a discrete domain and is assigned a vector of probabilities, labeled $Bel(v_i)$ (for $Belief(v_i)$). Each probability in $Bel(v_i)$ represents belief that v_i will take on a particular value. $D = (N, E)$ is a DAG such that a directed edge $e =< s_i, t_i >\in E$ indicates causal influence from source node s_i to target node t_i. For each node t_i, the strengths of causal influences from its parent s_i are quantified by a conditional probability distribution $p(t_i|s_i)$, specified in an $m \times n$ edge matrix, where m is the number of discrete values possible for t_i and n is the number of values for s_i. For complete coverage of Bayesian networks, their definition and use, updating algorithms, and complexity analysis, see [Pea88, Nea90, HMW95].

3.2 Modeling Software Uncertainties

There are many compelling reasons for using Bayesian networks for modeling software uncertainties: First, Bayesian networks offer a mathematically-sound computational model for uncertain reasoning. Also, their graph structure seems to match, at least conceptually, that of software systems. Thus, one should be able to construct a Bayesian model of a software system simply by annotating software artifacts with belief values representing developers' confidences and, correspondingly, software relations with conditional probability matrices (as is done here and in [ZR97a]). We note that the resulting network has to be sensible and sound (in the probabilistic sense), i.e., one should be able to interpret software relations as causal relationships among software artifacts. We also note that the notion of Bayesian belief corresponds to our earlier notion of degree of confidence. In the following, we use "belief" specifically to refer to a Bayesian value, whereas "confidence" is used more generally to indicate subjective assessment of a software entity.

It should also be noted that software artifacts, relations, and associated beliefs change frequently in all but the most trivial development processes. Bayesian networks may prove instrumental in capturing the dynamics of software change (with respect to confidence and uncertainty) by means of Bayesian updating. Furthermore, one's beliefs in software artifacts are typically influenced by many factors. This is easily accommodated in Bayesian networks since evidence from multiple sources can be combined to determine the probability that a variable has a certain value. Thus, Bayesian networks allow developers' confidences to be continuously updated during software maintenance.

Finally, we believe that by using Bayesian networks one can address real problems of software engineering, including, among others, effective hypertext navigation of large software spaces [ZR97b, ZO95], determining ultrahigh dependability of systems [LS93], and identifying performance bottlenecks and high-risk modules. Our choice of Bayesian networks, however justified, does not preclude the application of other techniques for modeling uncertainties. Instead, other approaches, including fuzzy, monotonic and non-monotonic logics, should be investigated and their relative strengths and weaknesses compared against those of Bayesian networks. This investigation is currently underway, its outcome to be reported in [Ziv97a].

4 Example: Uncertainties in CEquencer Software

4.1 CEquencer Requirements Uncertainties

In the following, we present the results of eliciting confidence levels or "belief values" from requirements analysts for the CEquencer system. To this end, the following steps were carried out:

- First we studied CEquencer requirements documentation, for example [Bec97]. Though the requirements for CEquencer were determined and specified prior to our study, we found the documentation commensurate with a REquirements BUilding process we have been using in research and classroom situations, called REBUS [SZH+91]. This enabled us to capture CEquencer requirements as a DAG of REBUS nodes, explained next.

- REBUS allows for software requirements to be captured as a hierarchical DAG of requirements elements. For CEquencer, the top levels of its requirements DAG are:

  1. The root node represents the entire CEquencer system.
  2. The main subcomponents of the system include Run (for running a method or sample table), Edit (for editing a method or sample table), Data analysis, Data management, and Help.
  3. Here, the main subcomponents above are decomposed into their respective subrequirements. The Run component is the most complex of the five, and includes File management, Direct control, Run menu operations, Tools, Log operations, and Replenish.

  Additional levels of decomposition exist, but space does not permit their inclusion here. Instead, a graphical depiction of the topmost levels of CEquencer's REBUS DAG, detailing Run requirements, is shown in Figure 1.

- Upon capturing the structure of requirements elements, probabilistic "belief values" for those elements were established. This was done by interviewing CEquencer's requirements analyst to record her confidence levels in those requirements. Her confidences are summarized in Table 3. Note that the third column of Table 3 cites those factors that have influenced and contributed to each belief. The notion of influential factors is discussed next.

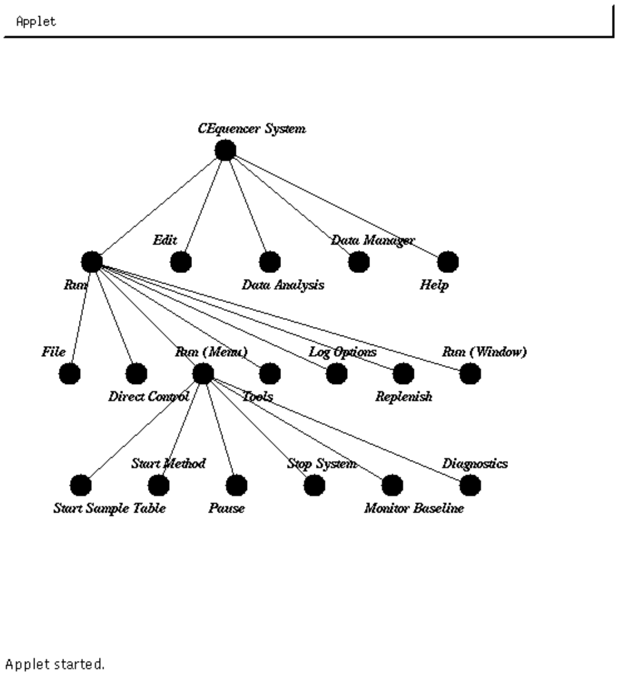

Figure 1: A REBUS DAG of CEquencer Requirements

4.2 Confidence Factors

In interviewing CEquencer's requirements analyst, it be bacame apparent that several factors affect her levels of confidence in requirements elements. We identify seven such Confidence Factors (CFs) below. Before listing the CFs, we make the following observations:

- CFs vary in degree of objectivity. For example, the analyst typically has more confidence in requirements that have already been addressed successfully in the past (e.g., in previous versions of CEquencer or similar Beckman projects). Based on this prior experience, the analyst more objectively obtains a higher level of confidence. On the other hand, when a requirements is less familiar, confidence levels regarding whether this requirement is likely to change or to be difficult to implement are attained more subjectively.

- CFs may increase or decrease developers' confidences. For example, a more complex requirement or one that is likely to change would tend to decrease confidence. On the other hand, the analyst would typically be more confident of requirements that are constrained, for instance, by the laws of physical and biological sciences.

- Multiple CFs may influence a single requirement, for instance, when a requirement is perceived as complex, likely to change, and restricted by laws of nature. The analyst's confidence level in this case is influenced by multiple factors and would decrease or increase accordingly.

We have identified CFs for CEquencer requirements. CFs are shown in Table 3, where the rightmost column is indexed by CF numbers, defined next.

1. **Incomplete or ambiguous requirements**. Developer or expert confidence in a requirements artifact tends to decrease if there exist concerns regarding the artifact's completeness (i.e., whether all aspects of the requirement are captured fully) and/or ambiguity (i.e., whether the requirement description is ambiguous).

2. **Anticipated changes**. CEquencer requirements, like those of most ongoing software projects, are in constant flux and undergo many changes. In cases where changes are anticipated or imminent for one or more requirements artifacts, expert confidence in those artifacts is, expectedly, reduced.

3. **Prior experience with a required capability**. Similar to most other systems, CEquencer designs and code modules are not developed from scratch. Rather, CEquencer improves upon as well as reuses elements of previously developed software. When existing components are reused, especially ones for which past experiences have been favorable, developer confidence is high. Conversely, when little or no experience exists, the degree of uncertainty increases and, correspondingly, confidence drops.

4. **Real world concerns and constraints**. These are problem domain issues that exist in the real world, specifically in CEquencer's physical operational environment. Generally, when laws of nature restrict operational behavior, expert confidence tends to increase.

5. **Hardware and firmware interface uncertainties**. These uncertainties typically arise at points of interface to hardware and firmware elements. An example is sending and receiving information to hardware components; this typically reduces confidence due to hardware uncertainties and, most importantly, interface uncertainties.

6. **Perceived complexity**. Requirements whose provision appears trivial or at least straightforward tend to increase developer confidence. Conversely, confidence is reduced for requirements that specify complex behaviors or interactions.

7. **Derived versus directly available data**. Is the information readily obtained from, for example, a device or an instrument, versus getting the information through some additional transfer and/or mathematical computation. There is typically additional uncertainty caused by computations and levels of indirection.

4.3 Bayesian Models of CEquencer Artifact Uncertainties

This section includes an example Bayesian network for two requirements elements, two test suites, and ten modules, together representing a small but meaningful "slice" of CEquencer. This slice is meaningful in that its constituent artifacts are largely reused or reworked from previous incarnations of CEquencer, thereby comprising a typical maintenance scenario. A Bayesian network was constructed for the following elements:

- Test suite *FSM-TP-Unit-Test-Plan* tests code modules *FsmStatus*, *FsmData*, *FsmTmProg*, and *FsmMethod* against requirements *Start-Method* and *Start-Sample-Table*. Similarly, test suite *FSM-Unit-Test-Plan* tests code modules *FsmRun*, *FsmConfig*, *FsmStatus*, *FsmData*, *FsmDirCtrl*, *FsmCmdResp*, and *FsmVxDIf* against the same two requirements elements.

- Confidence levels for requirements elements have been established earlier, in Table 3. Additional confidence levels for test artifacts (elicited from CEquencer testers), code modules (elicited from CEquencer programmers), and their relations were collected and are summarized in Table 2. Figure 2 is a graphical depiction of artifacts, their relations, and associated belief values. Figure 2 reflects a Java applet, developed in conjunction with the program in section 5 and made available on the World Wide Web [Ziv97b]. This applet was shared with CEquencer developers to both receive their feedback as well as enhance their understanding of and insight into artifact relationships.

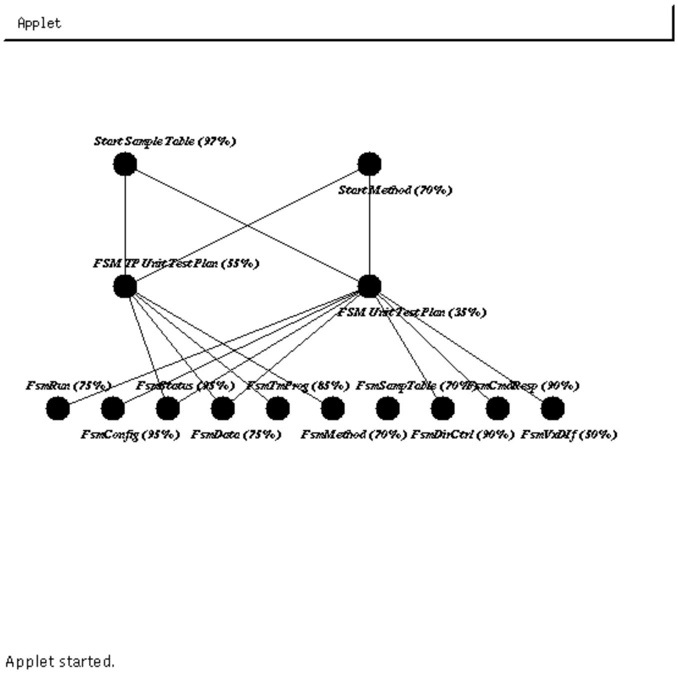

Figure 2: Example Bayesian-network Model for CEquencer Artifacts

5 Design and implementation

We used object-oriented (OO) techniques to design and implement Bayesian networks for software systems. Our design, shown schematically in Figure 3, includes an abstract base class *Graph*, itself an aggregate of two additional abstract classes, *Node* and *Edge*. Two classes, *Software System* and *Belief Network* are derived from *Graph*. Like *Graphs*, *Software Systems* are aggregations of *Software Artifacts* and *Software Relations*, while *Belief Networks* are aggregations of *Belief Nodes* and *Belief Edges*. Finally, using multiple inheritance, *Software Belief Networks* are derived from both *Software Systems* and *Belief Networks*; *Software Belief Nodes* from *Software Artifacts* and *Belief Nodes*; and *Software Belief Edges* from *Software Relations* and *Belief Edges*. This OO model is depicted in Figure 3 using Rumbaugh's Object Modeling Technique [RBP+91]. All classes and aggregation relations are shown but some obvious inheritance relations are left out to maintain clarity and readability.

The inheritance hierarchy of our model is arguably simple and may be elaborated further. A new class *Directed Graph* may be inserted between *Graph* and *Belief Network*. Also, class *Software Artifact* may be further specialized into kinds of software artifacts, such as requirements specifications, designs, test cases and test results; class *Software Relation* may be similarly specialized. These extensions, albeit useful, are beyond the scope of this paper; for our purposes, the model in Figure 3 is sufficient. Our design model is further simplified by a single-inheritance implementation. Multiple inheritance, though conceptually appropriate, may lead to implementation difficulties, including, among others, name clash resolution (cf. [Str94]). Indeed Java, the programming language chosen for implementation, allows multiple inheritance of interfaces only, but not of classes. Our Java program therefore implements a single inheritance scheme, from *Graph* to *Software System* to *Software Belief Network*.

Java was chosen for several reasons:

- It is an OO programming language, thereby offering smooth transition from an OO design

Artifact	Confidence/Belief Value	Elicited From
Start/ Run Sample Table	97%	Requirements Analyst
Start/ Run Method	70%	Requirements Analyst
FSM TP Unit Test Plan	55%	Run Module Tester
FSM Unit Test Plan	35%	Run Module Tester
FsmRun	75%	Run Module Programmer
FsmConfig	95%	Run Module Programmer
FsmStatus	85%	Run Module Programmer
FsmData	75%	Run Module Programmer
FsmTmProg	85%	Run Module Programmer
FsmMethod	70%	Run Module Programmer
FsmSampleTable	70%	Run Module Programmer
FsmDirCtrl	95%	Run Module Programmer
FsmCmdResp	90%	Run Module Programmer
FsmVxDIf	50%	Run Module Programmer

Table 2: Example Software Belief Network for CEquencer

model to implementation. Specifically, our conceptual notion that software systems and Bayesian networks share common attributes is still visible and traceable in the code.

- It offers several advantages over other OO languages, including dynamic code linking, automatic garbage collection, and support for creation and management of execution threads. Support for multiple threads is particularly appealing for Bayesian network implementation, since Pearl's algorithm can be implemented by message passing among network nodes, where each node executes in its own thread [1].

- Our implementation interacts with an existing implementation of Bayesian updating that was developed using Java and made available on the WWW, called JavaBayes [Coz96].

Our implementation allows for software artifacts, relations, and associated belief values, to be defined and entered. Bayesian updating, as implemented in JavaBayes, is then used for accepting new evidence and for propagating revised belief values throughout the software network. This implementation was used in [ZR97a] for Bayesian-network confirmation of developers' expectations for an elevator control system. Here, it is used for Bayesian-network modeling of requirements and testing uncertainties in CEquencer software. Additional modeling of Beckman software uncertainties is currently underway, to be reported elsewhere [Ziv97a].

6 Summary

In this paper we presented three related concepts:

1. *MUSE*: The Maxim of Uncertainty in Software Engineering, stating that uncertainties are abundant in software development.

2. *SBN*s: Software belief networks, combining software systems with Bayesian networks by annotating software artifacts and relations with belief values representing developers' confidences.

[1]Multiple threads are not supported in the current implementation.

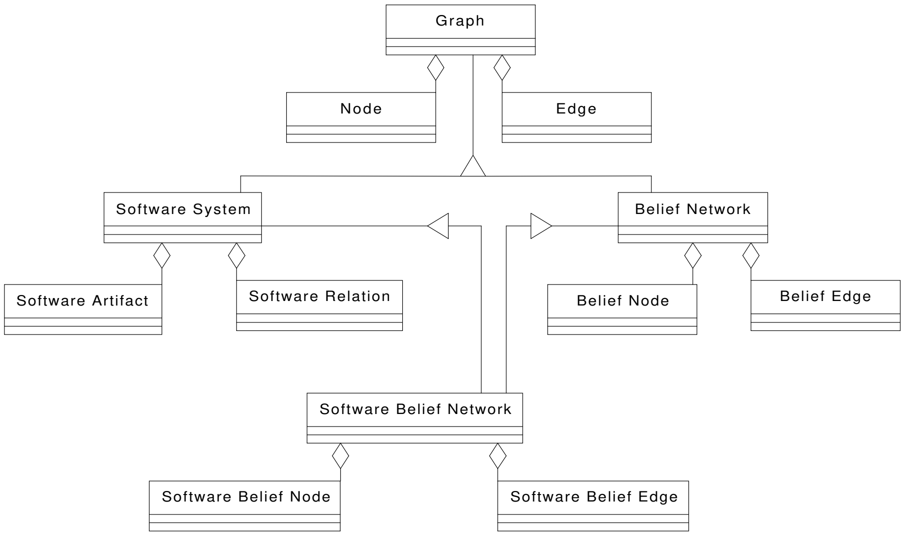

Figure 3: An Object Oriented Model of Software Belief Networks

3. *CF*s: Confidence Factors, which may decrease or increase developers' levels of confidence in software artifacts and relations.

We provided several examples of software uncertainties, constructed several SBNs, implemented SBNs in Java, and presented seven factors that have influenced levels of confidence in CEquencer requirements. Several impediments and limitations of our approach were observed, including the upfront cost of obtaining prior belief values, the need to ensure that software belief networks retain causality and variable independence, and the assumption that software developers, domain experts, and related project information are available and accessible.

More encouragingly, however, our preliminary experience with the Bayesian-network approach indicates that:

- The conceptual view of both software systems and Bayesian networks as interrelated "webs" of nodes and links seems to offer a convenient metaphor that also maps well into subsequent design and implementation. Specifically, early depictions of CEquencer artifact webs were developed as Java applets and placed on the World Wide Web (see [Ziv97b]); these applets were then viewed and reviewed by Beckman developers for accuracy and relevancy.

- The CEquencer system, like most other software, proved to be fraught with uncertainties, thereby confirming our suspicions for at least one real-life system. Specifically, CEquencer software embodies many, often subtle, problem domain uncertainties, including uncertainties stemming from laws of physics and chemistry in the software's operational environment as well as from vaguely defined boundaries between software versus hardware components.

- Finally, our notion of *software uncertainties* was well received by Beckman developers, offering convenient means in which to describe their confidences and beliefs regarding CEquencer software. This in turn contributed to increased “confidence” in our approach and our ability to model and manage additional uncertainties in the future.

We plan to further pursue Bayesian-network modeling of software uncertainties, especially with repsect to evolution and maintenance, where uncertainties are most abundant. Specifically, we wish to explore how Bayesian-network models may affect and guide future development steps.

References

[Bec97] Beckman Instruments. *CEQUENCE DNA Analysis System Software Requirements*, January 1997. Company Confidential.

[Coz96] Fabio Cozman. Javabayes version 0.2: Bayesian networks in Java, 1996. WWW Document http://www.cs.cmu.edu/ fgcozman/Research/JavaBayes/Home/index.html.

[CPRZ85] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A comparison of data flow path selection criteria. In *Proceedings of the Eighth International Conference on Software Engineering*, pages 244–251, London, August 1985. ACM SIGSOFT.

[CPRZ89] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, SE-15(11), November 1989.

[Cro93] Bruce W. Croft. Knowledge-based and statistical approaches to text retrieval. *IEEE Expert*, 8(2):8–12, April 1993.

[DN84] J. W. Duran and S. Ntafos. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering*, SE-10(4):438–444, July 1984.

[Fri88] Mark E. Frisse. Searching for information in a hypertext medical handbook. *Communications of the ACM*, 31(7):880–886, July 1988.

[FW88a] Phyllis Frankl and Elaine Weyuker. An Applicable Family of Data Flow Testing Criteria. *IEEE Transactions on Software Engineering*, SE-14(10):1483–1498, October 1988.

[FW88b] Phyllis G. Frankl and Elaine J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[FW93] Phyllis G. Frankl and Elaine J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering*, 19(3):202–213, March 1993.

[HMW95] David Heckerman, Abe Mamdani, and Michael P. Wellman. Real-world applications of bayesian networks. *Communications of the ACM*, 38(3), March 1995. Special Issue on Uncertainty in AI.

[HT90] Dick Hamlet and Ross Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, December 1990.

[Hum95] Watts S. Humphrey. *A Discipline for Software Engineering*. SEI Series in Software Engineering. Addison-Wesley, 1995.

[iAA96] Uncertainty in AI Association. Deployed bayesian-nets systems in routine use, October 1996. WWW Document http://www.auai.org/BN-Routine.html.

[JW89] Bingchiang Jeng and Elaine J. Weyuker. Some observations on partition testing. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 38–47, Key West, Florida, December 1989. ACM SIGSOFT. Published as *ACM SIGSOFT Software Engineering Notes 14*(8).

[LB96] Jacques-Louis Lions and The Inquiry Board. Ariane-5 flight 501 failure, July 1996. WWW Document http://www.esrin.esa.it:80/htdocs/tidc/Press/Press96/ariane5rep.html.

[LS93] Bev Littlewood and Lorenzo Strigini. Validation of ultrahigh dependability for software-based systems. *Communications of the ACM*, 36(11):69–80, November 1993.

[LT93] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[Nea90] Richard E. Neapolitan. *Probabilistic reasoning in expert systems: theory and algorithms.* Wiley, New York, New York, 1990.

[Pea88] Judea Pearl. *Probabilistic reasoning in intelligent systems: Networks of plausible inference.* Morgan Kaufmann Publishers, San Mateo, CA, 1988.

[Pre92] Roger S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw Hill, New York, New York, third edition, 1992.

[RAO92] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 105–118, Melbourne, Australia, May 1992.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1991.

[RC85] Debra J. Richardson and Lori A. Clarke. Partition analysis: A method combining testing and verification. *IEEE Transactions on Software Engineering*, SE-11(12):1477–1490, December 1985.

[Str94] Bjarne Stroustrup. *The C++ programming language.* Addison-Wesley, Reading, Mass., second edition, 1994.

[SZH+91] Stanley M. Sutton, Jr., Hadar Ziv, Dennis Heimbigner, Harry E. Yessayan, Mark Maybee, Leon J. Osterweil, and Xiping Song. Programming a software requirements-specification process. In *Proceedings of the First International Conference on the Software Process*, pages 68–89, Redondo Beach, CA, October 1991. IEEE Computer Society Press.

[WJ91] Elaine J. Weyuker and Bingchiang Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, July 1991.

[WWH91] Elaine J. Weyuker, Stewart N. Weiss, and Dick Hamlet. Comparison of program testing strategies. In *Proceedings of the Symposium on Software Testing, Analysis, and Verification (TAV4)*, pages 1–10, Victoria, British Columbia, October 1991. ACM SIGSOFT, ACM Press.

[Ziv97a] Hadar Ziv. *Bayesian-network Modeling of Software Uncertainties.* PhD thesis, University of California, Irvine, 1997. Working Title, In Preparation.

[Ziv97b] Hadar Ziv. Java applets for Beckman CEquencer software, March 1997. WWW Document http://www.ics.uci.edu/ ziv/java/bayesian_example.html.

[ZO95] Hadar Ziv and Leon J. Osterweil. Research issues in the intersection of hypertext and software development environments. In Richard N. Taylor and Joëlle Coutaz, editors, *Software Engineering and Human-Computer Interaction*, volume 896 of *Lecture Notes in Computer Science*, pages 268–279. Springer-Verlag, Berlin Heidelberg, 1995.

[ZR97a] Hadar Ziv and Debra J. Richardson. Bayesian-network confirmation of software testing uncertainties. Technical report, University of California, Irvine, January 1997. submitted to ESEC/FSE'97.

[ZR97b] Hadar Ziv and Debra J. Richardson. Lost and found in SoftwareSpace: A Bayesian approach. In John Tabor, editor, *Multimedia Technology and Applications (MTAC'97)*, Irvine, California, March 1997.

Requirement	Confidence/Belief Value	Related CFs
Run Module General Requirements	97%	2, 3
Restore Default Data Monitor Display	98%	3
Save Data Monitor Display	98%	3
System Preferences	85%	1
Load/ Unload Trays	50%	2, 5
Direct Control Tray position	50%	2, 5
Setting capillary temperature	80%	5
Denature Samples	85%	3
Denature Samples Tray position	50%	3, 5
Capillary Alignment	97%	3
Inject	95%	3
Injection Tray position	80%	2, 5
Separate	75%	2, 5
Separation Tray position	50%	2, 5
Gel Capillary Fill	60%	3, 5
Start/ Run Sample Table	97%	3
Start/ Run Method	70%	2
Configuration Confirmation	97%	3
Pause	85%	6
Stop System	70%	1, 2, 3
Monitor Baseline	95%	3
Diagnostics	45%	1, 2, 3, 6
Auto scale	98%	3
Pause Data	90%	3
Unzoom/ Unzoom all	98%	3
Display options	98%	3
Log options	90%	5
Freeze Log	80%	5
Capillary information	90%	5
Release/ Install Capillary Array	40%	3, 6, 7
Gel/ Buffer information	90%	5
Release Gel Cartridge	65%	3, 5
Run About Box (Help)	99%	3
Instrument information	95%	5
Status monitor	85%	2, 5
Activity	99%	3
Progress or Active	87%	3, 5
Method	90%	3, 5
Voltage	90%	3, 5
Total Averaged Current	70%	3, 5, 7
Gel level	70%	3, 5, 7
Gel life	98%	3
Capillary usage	80%	3, 5
Capillary life	75%	7
Device position	90%	3
Interpreting capillary temperature	70%	4, 5
Laser 1 Hours	97%	5
Laser 2 Hours	97%	5
Sample	98%	3, 5
micro-Amps	80%	3, 4, 5
Viewing data	98%	3, 5

Table 3: CEquencer Requirements, Beliefs, and Confidence Factors