Bayesian-network Confirmation of Software Testing Uncertainties

Hadar Ziv
Debra J. Richardson
Department of Information and Computer Science
University of California, Irvine
Irvine, California 92697-3425
{ziv,djr}@*ics.uci.edu*

Abstract

In this paper, we claim that software development will do well by explicit modeling of its uncertainties using existing uncertainty modeling techniques. This is accomplished initially by stating the Maxim of Uncertainty in Software Engineering (MUSE), followed by a detailed presentation of uncertainty in software testing. We then propose that a specific technique, known as Bayesian Belief Networks, be used to model software testing uncertainties. We demonstrate the use of Bayesian networks to confirm beliefs in the validity of software artifacts and relations in an elevator control system. We describe a prototype implementation that allows for such "software belief networks" to be defined and updated. We conclude with a discussion of issues, concerns, and future prospects for modeling software uncertainties.

Keywords: Uncertainty modeling, Bayesian networks, Software testing, Software maxims.

1 Introduction

Future prospects for software development appear promising. New areas of rapid growth include hypertext and multimedia systems for the World Wide Web (WWW), object-oriented methods and distributed object technology, software patterns and reusable class libraries. These prospects, albeit exciting, are clouded by the uncertainties and associated risks of developing complex software. Perhaps the most alarming manifestation of software risks are software–related accidents, such as described for the Therac-25 medical radiation device [LT93] or else reported, for example, in the *comp.risks* newsgroup. In addition to software–related accidents, software risks and uncertainties also carry the blame for known symptoms of the software crisis, including high development costs and unplanned schedule slips. Consequently, researchers and practitioners alike spend considerable efforts attempting to manage software uncertainties and relieve associated risks.

While software risk management is an active research area that has over time produced significant results (cf. [Boe89, Boe91]), software uncertainties remain largely unexplored (with notable exception of software reliability models [Lit79] and validation of software dependability [LS93]). This is particularly surprising in light of extensive research in modeling and management of uncertainties carried out in various artificial intelligence (AI) domains.

In this paper, we claim that software development will do well by modeling its uncertainties using existing uncertainty modeling techniques. The purpose of uncertainty modeling in software engineering is twofold:

1. Since uncertainties permeate software development, a software model offering explicit representation of those uncertainties would be more accurate and realistic than one where uncertainties are disregarded.

2. By modeling its uncertainties, certain structural and behavioral aspects of a software system become more visible and understandable, thereby enabling future development steps to be carried out more efficiently and effectively.

Clearly modeling all possible uncertainties for all of software development using all possible modeling techniques is impractical. Although we plan to extend our results broadly in future efforts, we focus here on uncertainties that arise during software testing and maintenance; we choose an uncertainty-modeling technique known as Bayesian Belief Networks; and we select a single software development project as case study.

1.1 Key Ideas and Contributions

The key ideas and contributions of this paper are as follows:

1. Recognizing the abundance of software uncertainties and the need for uncertainty modeling and management: This is accomplished initially by presenting the Maxim of Uncertainty [1] in Software Engineering, or MUSE [ZRK96], as follows: "Uncertainty is inherent and inevitable in software development processes and products."

2. Proposing that uncertainty modeling techniques be used to model uncertainties for specific software situations: Though we firmly believe that uncertainty modeling benefits many software development tasks, here we focus on its application to software testing and maintenance concerns, specifically error discovery and defect detection. We identify three broad categories of applying uncertainty modeling to software engineering tasks, as follows:

[1] A "maxim" is an expression of general truth or principle [GR95].

(1) Confirmation: Certain characteristics and behaviors of software systems would seem "sensible" to most developers and therefore expected to hold true. Confidence in the correct behavior of a code module, for example, 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 is done in this paper). Models for confirmation 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, established test suites for regression testing that are based on one or more test adequacy criteria (cf. [CPRZ89]). Uncertainty may still exist, however, regarding the true defect-detection ability of given test suites for given systems. This uncertainty may be modeled using uncertainty modeling techniques; the resulting model may in turn be used to evaluate the test suite's ability to detect defects.

(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.

3. Conducting case studies on the applicability of uncertainty modeling, specifically using Bayesian networks, to real-life software situations. These case studies may be organized by the categories defined above: Confirmation, evaluation, and prediction.

   (1) Confirmation: Here, Bayesian networks are used to model software testing uncertainties for an elevator system example. This constitutes a *confirmation* case study.

   (2) Evaluation and Prediction: In an upcoming paper (see [ZR97]), we present case studies for *evaluation* and *prediction* applied to software developed by Beckman Instruments in Fullerton, California; this software 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.

4. Implementing Bayesian-network capabilities for software systems. We describe our design and implementation of a system that allows to define software systems as networks of interrelated artifacts with associated belief values; these "software belief networks" are later subject to Bayesian updating (originally defined by Pearl [Pea88]). We employ an existing implementation of Bayesian updating available on the WWW.

1.2 Organization

This paper is organized as follows: First, examples of uncertainty in select domains of software development are presented, followed by three common sources of uncertainty in software engineering. We then present the Maxim of Uncertainty in Software Engineering, followed by detailed discussions of uncertainty in software testing, including test planning, test enactment, error tracing, and quality estimation. We describe a specific technique for uncertainty modeling and management, called Bayesian belief networks. We demonstrate the use of Bayesian networks to confirm beliefs in the validity of software artifacts and relations developed for an elevator control system. Our implementation of Bayesian networks for modeling uncertainty in software systems is then described.

We conclude with a discussion of issues, concerns, and future prospects for modeling software uncertainties, both specifically using Bayesian networks and in general.

We note that uncertainty is evident in most of software development, as well as in other fields of science and engineering and in everyday situations (see [Ste95] p. 460, for examples of everyday uncertainties). Detailed exposition of uncertainty in general is beyond the scope of this paper. We nevertheless hope that this paper serves as foundation for future broad discussions of software engineering uncertainties. While reading this paper, the reader is encouraged to consider occurrences and impacts of uncertainty in her own domains of interest and expertise.

2 Software Uncertainties

2.1 Uncertainty in Software Engineering Domains

Here, we present four select domains of software engineering where uncertainty is evident. Later, we discuss uncertainty in software testing in greater detail. We list frequently asked questions for each domain and indicate associated uncertainties. We note that these questions often require answers of degree instead of binary yea or nay. Consequently, we later propose that such questions are best addressed by means of probability values.

2.1.1 Uncertainty in requirements analysis

Successful software development is often hindered by the generally poor state of most requirements descriptions. Software requirements analysis typically include 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." Analysis of software requirements therefore inevitably introduces uncertainties, including: 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?

2.1.2 Uncertainty in the transition from system requirements to design and code

Software development typically requires the system to be represented at multiple levels of abstraction, including, for example, requirements analysis models, 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?

2.1.3 Uncertainty in software re-engineering

Software re-engineering includes reverse engineering of an existing system into higher-level architectural descriptions, followed by forward engineering of a revised system implementation [ZKR96, GKM95]. Thus, in addition to forward-engineering uncertainties, software re-engineering also includes the following uncertainties: How well does available system documentation correspond to program source code? How accurately do models that were reverse-engineered from program source code represent domain abstractions [GKM95]? To what degree can original system documentation be used in the reverse engineering process? For a detailed exposition of uncertainties in reverse engineering and re-architecting, see [ZKR96].

2.1.4 Uncertainty in software reuse

Uncertainty in software reuse has not only been recognized but also addressed by many researchers, for example Prieto-Diaz and Arango [PD91a, PD91b]. Special sessions on uncertainty in software reusability were held at the 1994 and 1996 IPMU conferences, including papers on, among others, uncertainties in the composition of reusable components [MW96] and in object recovery [GK94]. Effective reuse of software components introduces several uncertainties, including: How to specify reusable-module interfaces completely and sufficiently? What is one's confidence that an existing (i.e., available for reuse) component meets one's usage needs? And finally, given a reusable component, how can it be tailored to existing project constraints and assumptions?

2.2 Sources of Uncertainty in Software Engineering

2.2.1 Uncertainty in the problem domain

Software development typically requires that one or more models of the application domain be built. Uncertainties are abundant in real-world domains; consequently, a software system based on real-world models would inevitably embody domain uncertainties. Such domain uncertainties, unfortunately, are often only hinted at or simply ignored in the system's domain model. This may in turn lead to discrepancies between domain reality and system assumptions, which may ultimately lead to software risks and uncertainties. In embedded software systems, additional uncertainties exist due to the interaction of external software, hardware, and mechanical components. Ignoring these uncertainties may be hazardous, possibly fatal [2].

2.2.2 Uncertainty in the solution domain

Software systems constitute solutions to real world problems. Software solutions may introduce additional uncertainties above and beyond those attributed to the problem domain. A known example of solution domain uncertainty occurs in concurrent-program debugging. The key difficulty in debugging concurrent programs is due to race conditions, which introduce uncertainty, where erroneous program behavior observed in one execution may not be evident in subsequent executions [MH89]. Moreover, any attempt to "probe" the program for additional information, for example by instrumenting it, may adversely affect the probability of reproducing the erroneous behavior. This phenomenon is often referred to as the "probe effect" [Gai85, Gai86], where by attempting to observe program behavior, software probes may inadvertently affect the outcome of the observation.

2.2.3 Human participation

Human action is intrinsically inexact and susceptible to errors, its outcome inherently uncertain. Since software is largely inspired and developed by humans, uncertainty and unpredictability are inevitable throughout the software life cycle. That software development is still largely human-intensive may seem trivial, yet surprisingly few methods offer explicit modeling of the inexact

[2] Leveson and Turner [LT93] report on the Therac-25 accidents, where the possibility of software–related uncertainties was dismissed and disregarded by the manufacturer.

and uncertain nature of human involvement and its implications. Rule-based formalisms for software process modeling, for example, such as Marvel/Oz [HKBBS92] Merlin [PS92], and Articulator [MS91] allow for process steps to be captured as rules, but associated uncertainties cannot be modeled.

3 The Maxim of Uncertainty in Software Engineering

Software is often developed in the presence of daunting domain complexities and under circumstances that are uncertain, incomplete, or otherwise vague. Software systems appear unconstrained by any laws of nature, maxims of behavior, or principles of construction. It has long been recognized, however, that software engineering would do well by a standard set of guidelines, maxims and principles. Software maxims and principles have been identified by Davis [Dav95a], Ghezzi [GJM91], Brooks [Bro75, Bro87], Goldberg and Rubin [GR95], and others. In this paper we state the Maxim of Uncertainty in Software Engineering (MUSE) as follows:

Uncertainty is inherent and inevitable in software development processes and products.

We also propose the following corollary to the maxim:

Software uncertainties should be modeled and managed explicitly.

Similar to other maxims and principles of science and engineering, MUSE is defined generally and abstractly, is applicable to a wide range of practical software development situations, and can be witnessed and substantiated repeatably and predictably. MUSE should still, however, be applied judiciously and appropriately.

3.1 Applying The Maxim

Software engineering processes and products include elements of human participants (e.g., designers, testers), information (e.g., design diagrams, test results), and tasks (e.g., "design an object-oriented system model," or "execute regression test suite"). Uncertainty occurs in most if not all of these elements. A software modeling activity would therefore do well to apply MUSE by explicitly modeling one or more uncertainties, taking into account the following concerns:

- What is the goal of the software modeling activity?
- When and why is uncertainty modeling relevant?, and
- What approaches or formalisms for uncertainty modeling are employed?

4 Software Testing Uncertainties

4.1 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.

- Source code, produced by coding and debugging.

According to MUSE, uncertainty permeates these processes and products. Plans to test these artifacts, therefore, will carry their uncertainties forward.

Software testing, like other development activities, is human intensive and thus introduces uncertainties. These uncertainties may 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.

4.2 Test Enactment

Test enactment includes test selection, test execution, and test result checking. Test enactment is inherently uncertain, since only exhaustive testing in an ideal environment guarantees absolute 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, discussed next, that introduce uncertainties to test enactment activities.

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 represens one's belief in its ability to detect defects. An example of assigning confidence values to path selection criteria is given below.

Test execution involves actual execution of system code on some input data. Test execution may still include uncertainties, however, as follows: the system under test may be executing on a host environment that is different from the target execution environment, which in turn introduces uncertainty. 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.

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. Specification-based oracles instill the highest confidence, but still include uncertainty stemming from discrepancies between the specification and customer's informal needs and expectations.

We have modeled test oracle uncertainties for an extended test scenario for an elevator system, but space does not permit inclusion of the entire model in this paper. Instead, we present two snippets from the complete scenario: One, described next, is in the domain of path selection criteria. Then, in section 6, a unit test scenario is used to demonstrate confirmation of one's common-sense expectations.

4.3 Example: Path Selection Testing Criteria

In [CPRZ85, CPRZ89], the authors present a subsumption hierarchy that imposes a partial order on different data flow path selection criteria with respect to their ability to provide adequate coverage of a given program. The subsumption relationship implies relative strength of criteria, which may in turn be recast in terms of confidence levels, as follows: if criterion A subsumes criterion B, then A is "stronger" than B, and therefore one would have more confidence in A's defect detection abilities than in those of B[3]. 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] is shown in Table 1.

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

Though plausible, Table 1 still raises some important questions, including:

1. Why are confidence values relatively low?

2. How are confidence values determined?

3. How are confidence values used?

Questions (2) and (3) above are best addressed after additional details are provided (see section 8). As for question (1), we observe that low confidence values indicate that even "strong" path selection criteria do not necessarily incur high levels of confidence in their defect detection abilities. This is because path selection does not take into account, for example, data value selection. Some defects are only revealed by particular data values, but not by others. Low confidence values for path selection criteria therefore reflect the criteria's inability to guarantee defect detection.

4.4 Quality Estimation

Software testing is instrumental in establishing quality and high assurance in software processes and products. A key concern of software testing is "When to stop testing?", which is often answered

[3]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.

by means of quality estimation. We consider reliability testing and reliability growth modeling to be among the most mature techniques for software quality assessment [LS93] and therefore focus on them below.

Considerable work in software reliability modeling is based on a probabilistic notion of uncertainty. A probabilistic model of software behavior is needed since neither program testing nor formal proof of program correctness can guarantee complete confidence in the correctness of a program [Goe85]. Software reliability measures to what degree a software system behaves as expected, thereby modeling system behavior as observed by its users, as opposed to static or dynamic properties of the code itself. Examples of measures used in software reliability include frequency of failure and mean time to failure. Software reliability may therefore be defined as the probability that software faults do not cause a program failure during a specified exposure period in a specified use environment [Goe85].

Hamlet [Ham96] and Littlewood [LS93] extend existing reliability theory by defining "software dependability" as a statistical measure of software quality. Specifically, Hamlet uses Blum's notion of self-checking programs [BW94] to define the dependability of program P at input X as the confidence probability that P is correct (with respect to its specification) at X.

Software reliability models demonstrate that software uncertainties may be measured and therefore captured explicitly. Furthermore, reliability modeling may also be used to estimate future software quality. When predicting future reliability, one often assumes that software systems are used with statistical regularity. This assumption, however, introduces additional uncertainty, since future users may exhibit vastly different usage patterns. We conclude that probabilistic measures of software reliability may provide initial estimates of confidence (i.e., *prior probabilities*) for software artifacts and relations. This is discussed further below.

4.5 Error Tracing

When a software failure is detected, the source of the error must be found. The error may have been introduced at an early stage of development, such as requirements analysis or system design, or later during coding. Effective error tracing, also known as the "discovery task" [DBSB91], requires that software artifacts are interrelated among themselves as well as to informal customer requirements.

Software traceability is the creation, management, and maintenance of relations from one software entity to other entities [Dav95b]. It is often assumed that software traceability is afforded by software engineering environments (SEEs) that offer software tool integration, object management systems, and hypertext capabilities in a heterogeneous environment (cf. [Kad92]). For a large network of software artifacts and relations, however, traceability is still hampered by the cognitive difficulty of sifting through large volumes of interrelated information. Software engineers are likely to get disoriented in large software spaces due to uncertainties encountered during navigation, such as "Where am I?", "How did I get here?", and "Where can I go next?" [ZO95].

We conclude that explicit modeling of uncertainty is relevant and applicable to many software engineering situations and may help ameliorate practical problems, such as effective navigation in large software spaces.

5 Modeling Uncertainty

We suggest that uncertainties associated with one or more properties of software artifacts be modeled and maintained explicitly. We propose that one or more Bayesian networks be constructed for a software system to model its relevant uncertainties. Once constructed, the Bayesian model may be used for purposes of confirmation, evaluation or prediction for the underlying software system.

One or more questions of interest, such as identification of high risk or error prone components, can be answered using the Bayesian model. In some cases, a complex software system can be visually navigated more effectively by using the Bayesian model to guide the software engineer to artifacts that are more likely to exhibit some property. We now describe the Bayesian approach to uncertainty modeling.

5.1 Bayesian Belief Networks

Bayesian belief networks have been used in artificial intelligence research to provide a framework for reasoning under uncertainty [Pea88, Nea90]. Bayesian networks have been used extensively in a wide range of applications [HMW95], including medical diagnosis, price forecasting, future crop production, automated vision, sensor fusion, and modeling of manufacturing processes. Of particular relevance are successful applications of the Bayesian approach 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].

Informally, a Bayesian network is a graphical representation of probability relationships among random variables. A Bayesian network is a Directed Acyclic Graph (DAG), where graph nodes represent variables with domains of discrete, mutually exclusive values. 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 of a Bayesian 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 $Belief(v_i)$ or $Bel(v_i)$. 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.

The structure of a Bayesian network is usually determined by consultation with experts. Probabilities in edge matrices can either be estimated by experts or compiled from statistical studies. An important assumption of Bayesian networks is variable independence: a variable is independent (in the probabilistic sense) of all other non-descendant variables in the network except 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 7, an updating scheme based on a message passing metaphor is conducive to distributed implementation.

Pearl's updating algorithm requires that two additional vectors, labeled λ and π, be used. λ vectors are used to send messages up the network, from a child node to its parent. λ values are typically set to one initially [4], before any evidence is propagated, but are later revised to reflect new evidence. When new evidence is observed, for example, if "test suite T detected a defect in code unit M," then the corresponding λ vector is revised to $\binom{1}{0}$ or $\binom{0}{1}$ as appropriate. Revised λ

[4] Unlike *Belief* and π vectors, values in the λ vector do not need to sum to one.

values are sent as a message up to the revised node's parent and multiplied by the edge matrix. The resulting vector is multiplied by the parent node's λ vector to yield a new λ. This upward propagation repeats until the network's root node is reached. Similarly, downward propagation proceeds by means of messages, indicated by π vectors, sent from a parent node to its child, until belief values for all network nodes are updated.

Regarding computational complexity, Bayesian updating of an arbitrary network (i.e., where cycles may exist in the underlying undirected graph) is known to be NP–hard [Coo90]. If, however, the network is tree-structured, Pearl's updating algorithm is quadratic in the number of values per node and linear in the number of children per parent. For clarity and simplicity, we limit our discussion in this paper to tree-structured software networks. We wish to note, however, than many additional Bayesian updating algorithms exist that are polynomial in time and space. For comprehensive descriptions of Bayesian updating algorithms, see [Pea88, Nea90, HMW95].

5.1.1 Why Bayesian Networks?

We identify compelling reasons for using Bayesian networks for modeling uncertainty in software engineering. First, it is a mechanism to apply MUSE in practice, i.e., Bayesian networks provide a mathematically sound technique for explicit modeling of uncertainties inherent in software development. Moreover, their graph structure matches that of software systems. Thus, it is possible to impose a Bayesian network on a software system by associating belief values with artifacts and conditional probability matrices with relations. 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. In addition, since more than one belief value may be associated with a single software entity, multiple Bayesian networks can be imposed on a single software system.

Also, a software development process is highly dynamic in that software artifacts, relations, and associated beliefs are modified frequently. Bayesian networks are able to reflect dynamic changes in a software system 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. Finally, we believe that by using Bayesian networks one can address real problems of software engineering, including, among others, effective navigation of large software spaces, deciding when to stop testing, and identifying bottlenecks and high-risk components.

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 [Ziv97].

6 The Elevator System Example

A large software integration and demonstration effort of the Arcadia research project [Kad92] included development of software artifacts and relations for an elevator control system. The elevator example is a classic problem of software engineering and has been studied extensively, particularly in conjunction with formal specification languages [RAO92]. The problem is concerned with the logic required to move elevators between floors according to specified functional requirements as well as safety, liveness, and fairness constraints.

Software artifacts in the Arcadia elevator solution include functional decomposition of requirements developed using REBUS [SZH+91]; formal specification, including model-based specifications in Z [Spi89]; interval logic specifications using RTIL and GIL tools [DKMS+92]; object-oriented design diagrams; code modules implemented in Ada; and test suites, test criteria, and test oracles developed using TAOS [Ric94]. These elevator system artifacts are interrelated by means of software-artifact relationships.

We used Bayesian networks to model uncertainties in the elevator example. Probabilistic belief values, determined by consultation with domain experts, were assigned to software artifacts and relations. Though belief values were assigned and Bayesian updating was carried out for the entire elevator solution, space is too limited to show the complete example. Instead, for clarity and brevity, we demonstrate the Bayesian approach for code unit testing modeled by a subnetwork of only four elevator system artifacts. The elevator example is described in its entirety in [ZRK96].

6.1 The Unit Test Scenario

In the unit test scenario, a software entity is considered valid if it meets customer needs and expectations (cf. [GJM91]). Thus, valid design representation is traceable to requirements, valid code units successfully implement designs, etc. Given this definition, complete confidence in validity is clearly hard (if not impossible) to achieve in practice. Instead, a probabilistic belief value is associated with the statement "this entity is valid," and assigned to the corresponding entity.

Network node D represents a design specification element, say *Elevator_Controller_Interface_Spec*, in the unit test scenario. The probability $Bel(D)$ represents prior prior belief that D is valid. Similarly, node M represents an Ada code module, say *Elevator_Controller_Package*, in the unit test scenario. M is assigned a probability $Bel(M)$, representing prior belief that M is valid. There exists a causal relationship, M *implements* D, indicated by a directed edge $< D, M >$ in Figure 1.

In addition, test nodes $T1$ and $T2$ represent two test suites, corresponding to two different test selection criteria, say *All–Edges* and *All–Uses*. Test suites are executed against implemented code units and may succeed or fail. Test suite execution succeeds when no defects are detected, i.e., actual test results match expected results. Expected results for test result checking are provided either manually or by a test oracle. Here, a code module becomes invalid whenever a single defect is detected, i.e., if execution of one or more related test suite fails; consequently, the belief value assigned to the corresponding node is set to zero [5]. Note, however, that successful test execution does not instill complete confidence and, consequently, merely increases one's confidence in the module's validity. Expected increases and decreases in confidence values for elevator artifacts are confirmed in the unit test scenario by way of Bayesian updating, as reported below.

6.1.1 Initial State of Bayesian Network

We begin with design node D. Confidence in the validity of design specifications varies considerably among different projects, different design methods, and different designers. In the unit test scenario, D's prior belief value is determined to be .7. This is recorded in D's belief vector.

As shown in Figure 1, a π vector, used later for downward propagation, is also associated with D. Since new evidence is yet to be propagated, D's π vector is initially identical to $Bel(D)$. Similarly, since no propagation has occurred yet, D's λ values are initially set to 1.

A directed edge $< D, M >$ indicates that M implements D. Conditional probabilities in the corresponding edge matrix represent beliefs that M is valid (or invalid) given that D is valid (or invalid). Here, conditional probabilities were determined by domain experts as follows: when D is

[5] Alternate definitions of validity are discussed in section 8.

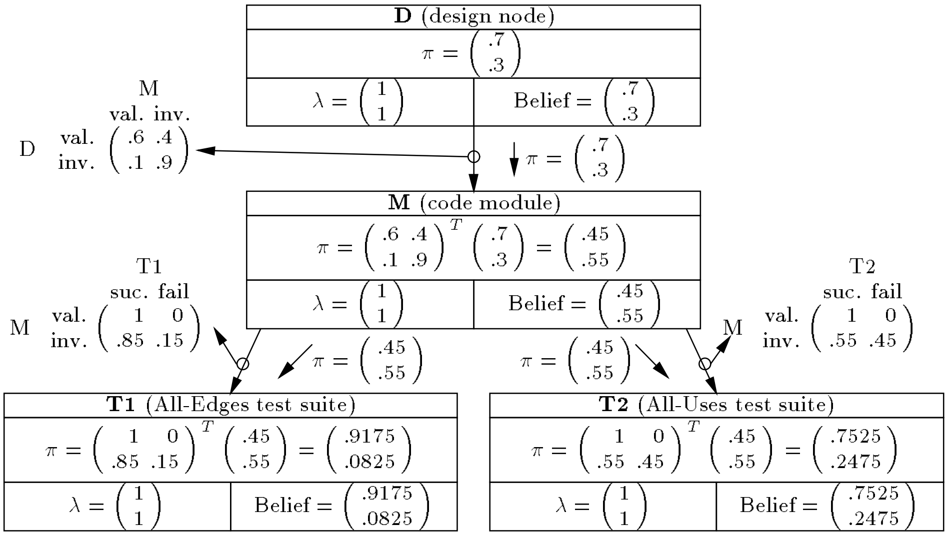

Figure 1: Initial Belief Network for Unit Test Scenario

valid, M is also valid with probability .6; M is invalid with probability .4. If D is invalid, however, then M is valid with only .1 probability and invalid with .9. These belief values are recorded in the edge matrix for $< D, M >$, shown in Figure 1.

Next, downward propagation is used to compute M's *Belief* vector. A π vector is computed for M by multiplying D's π vector (the downward message) by the transpose of the edge matrix for $< D, M >$. Resulting π values are then assigned to *Bel(M)*. Here, they indicate initial belief of 45% in M's validity, as shown in Figure 1.

Test suite $T1$ represents *All–Edges*, a relatively weak testing criterion in the subsumption hierarchy of [CPRZ89]. Table 1 indicates a confidence value of .15 in *All–Edges*'s defect detection abilities. Given that successful test suite execution implies no defects were detected, the following conditional probabilities were determined for the edge $< M, T1 >$: when M is invalid, $T1$ succeeds with probability .85 and fails, correspondingly, with .15. If M is valid, however, then $T1$ must always succeed; therefore, $T1$ succeeds with probability 1 and fails with probability 0, as captured in Figure 1.

Similarly, test suite $T2$ represents *All–Uses*, a stronger testing criterion. Table 1 associates a confidence value of .45 with *All–Uses*'s defect-detection abilities. This again determines the corresponding probabilities for $T2$ when M is invalid to be .45 and .55, respectively. The resulting edge matrix for $< M, T2 >$ is shown in Figure 1.

Next, belief values for $T1$ and $T2$ are computed, as before, by means of downward propagation. π values for $T1$ and $T2$ are computed by multiplying edge-matrix probabilities by a π message from M. Figure 1 shows the resulting belief values for $T1$ and $T2$, as follows: if M is invalid, $T1$ succeeds for M with probability 91.75% and $T2$ succeeds with 75.25%. These belief values correspond to one's higher confidence in the defect detection abilities of "stronger" path selection criteria.

6.1.2 Network State After Executing $T1$

Figure 2 illustrates the effects on the network of successful test execution of $T1$. Successful test suite execution provides new evidence; this evidence is recorded initially at $T1$ and then propagated throughout the network by way of Bayesian updating. Bayesian updating proceeds by means of sending upward and downward messages, as follows:

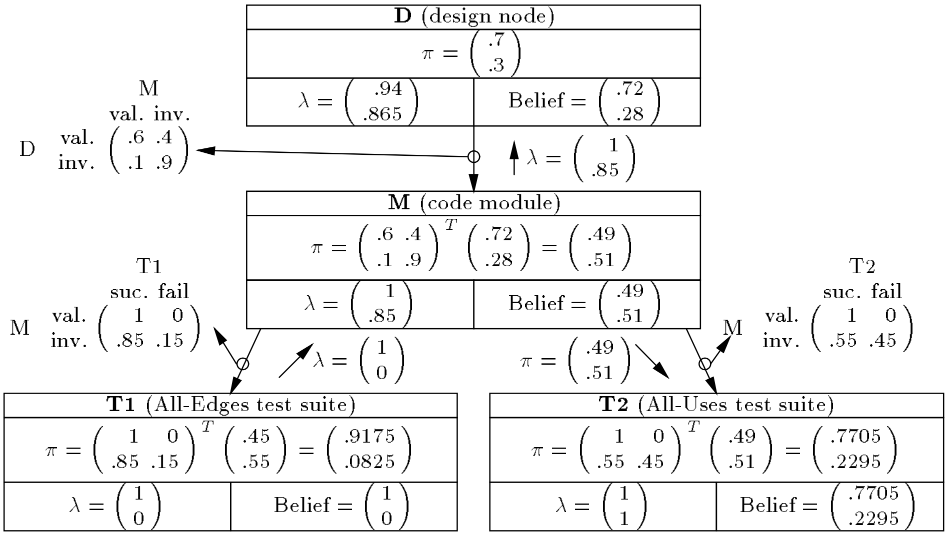

Figure 2: Belief Network After Successful Execution of $T1$

- $T1$'s λ vector is revised to $\binom{1}{0}$;
- $T1$ sends $\binom{1}{0}$ as a λ message to M, where it is multiplied by the edge matrix (without transposing);
- The resulting vector is then multiplied by M's current λ vector, yielding M's new λ.
- Next, M's *Belief* vector is revised by multiplying the current *Belief* vector by the new λ and normalizing the result such that the probabilities add to 1. As shown in Figure 2, this yields a revised belief of .49 that M is valid. This result is consistent with one's expectations that confidence in M should increase upon successful test execution.
- M sends an upward λ message to D, which is then used in revising D's λ and *Belief* vectors. Here, the revised belief value for D is .72, again confirming one's expectations that it should increase.
- M also sends a downward π message to $T2$. The π values are identical to M's new belief values established above, namely .49 and .52.
- $T2$ then recomputes its own π and belief vectors, yielding the network in Figure 2. The revised belief values for $T2$, .77 and .23, indicate an increase in our belief that execution of $T2$ for M will succeed. This is consistent with evidence that $T1$ executed successfully for M.

6.1.3 Network State After Executing $T2$

Next we consider the effects on the network of executing $T2$ (*All-Uses*). Upon execution of $T2$, belief values are revised, as before, by means of propagation and recomputation of λ and π values. If $T2$ were to fail, a defect has been detected and M is determined to be invalid. Specifically, $T2$'s λ vector is set to $\binom{0}{1}$ upon failure, and, after multiplication by the edge matrix and M's previous λ vector, M's belief vector is also updated to be $\binom{0}{1}$. This result confirms our expectation, discussed earlier, that code modules are invalidated by a single detected defect. Subsequently, additional upward propagation from M to D results, consistent with expectations, in decreased belief of .546 in D's validity.

If $T2$ succeeds, however, then Bayesian updating should indicate further increase in our confidence in the validity of both M and D. Here, M's belief value increases to 80.5%, while D's belief value increases to 85%. This increased confidence is, again, consistent with expectations. Due to space considerations, corresponding diagrams are not shown, but are identical in structure and appearance to those of Figures 1 and 2.

Note that increased confidence in software entities during testing may be compared against preset "confidence thresholds." When confidence thresholds are exceeded, a testing milestone – such as successful completion of regression testing – may have been reached.

7 Design and implementation

We used object-oriented (OO) techniques to design and implement software belief networks. Our design includes an abstract base class, *Graph*, modeled as an aggregation of two additional abstract classes, *Node* and *Edge*. From these abstract classes we derive *Software Systems* as aggregations of *Software Artifacts* and *Software Relations*; similarly, *Belief Networks* are defined as aggregations of *Belief Nodes* and *Belief Edges*. Finally, multiple inheritance is employed in deriving *Software Belief Networks* 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*.

The OO model described above is arguably simple and may be extended. Classes *Directed Graph* and *Hypertext Graph*, for example, may be added to the inheritance structure. Similarly, classes *Software Artifact* and *Software Relation* should be further evolved into specific kinds of software artifacts and relations. For the purposes of this paper, however, our model is sufficient. Our 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. Java was chosen for several reasons:

- It is an OO programming language, thereby offering smooth transition from our OO design model to implementation.
- 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 [6].
- Our implementation interacts with an existing implementation of Bayesian updating that was developed using Java and made available on the WWW.

Our implementation allows for software artifacts, relations, and associated belief values, to be defined and entered. Bayesian updating is then used for accepting new evidence (for example, test suite execution results) and for revising and propagating belief values throughout the software network. Our Java code implements a single inheritance hierarchy, from *Graph* to *Belief Network* to *Software Belief Network*, and similarly for *Software Belief Node* and *Software Belief Edge*.

The Java implementation was used to confirm the quantitative results of the elevator unit test example, as reported in section 6. This concluded our case study in Bayesian-network confirmation

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

of software uncertainties. We are currently carrying out Bayesian-network evaluation and prediction tasks for the Beckman software, to be reported elsewhere.

8 Discussion

8.1 How are belief values interpreted?

Bayesian belief values are typically (cf. [HMW95]) associated with observable phenomena that is described using binary True/False statements. When modeling everyday situations, for example, the prior belief value of "It is sunny" may be determined to be .9, while one's belief that "The dog is barking" may be .55 [Cha91]. Such statements are therefore viewed as observations on some entity's state, quality or property. Thus, values in a Bayesian network represent beliefs that an entity is in some state or possesses some quality or property.

Similarly, a single belief value associated with a single software artifact represents belief that the artifact is in some state or possesses some quality or property. In the unit test scenario, for example, the observed quality for design and code nodes is validity, whereas a test suite can be in one of two states, "success" or "failure."

In general, however, software artifacts may possess many different qualities, for example, correctness, robustness, reliability, safety, maintainability, and efficiency. They can also be in one of many different states. This implies that multiple Bayesian models may be associated with a single software network. It also implies that assignment of belief values to artifact qualities must be consistent with causal relationships in the network. The software belief networks in Figures 1 and 2 are "causally consistent."

8.2 When does a belief value become zero?

The elevator example demonstrates that belief values may be set to zero under certain conditions. A belief value of zero may have significant implications for other belief values because of Bayesian updating. Determining whether a belief value should be zero is therefore important as well as potentially difficult. This decision is influenced, for each belief value, by the quality of the associated entity.

In the elevator unit test scenario above, for instance, Bayesian values represent beliefs that source code units are "bug free" or otherwise valid with respect to specified requirements. In this case, the failure of a single test suite must cause the belief value to be set to zero. It is also conceivable, however, that test oracles used for test result checking are themselves suspect. In this case, one has only limited confidence in the testing process itself, and, consequently, failed execution of a test suite does not imply a belief value of zero. Furthermore, other scenarios, for example one where complex software includes many modules and is developed under stringent schedule constraints, may allow for code units to contain known defects under certain circumstances. In this case, uncertainty is modeled for a quality other than program correctness, say "acceptability." Belief values for program acceptability should decrease with each failed execution of a test suite, but do not necessarily become zero upon single failure. Belief should only become zero when, for example, a preset threshold (e.g., maximum number of defects allowed) is exceeded.

8.3 Where do belief values come from?

To use Bayesian networks, one must specify prior belief values for network nodes as well as conditional probabilities for causal influences. Certain independence assumptions hold, as mentioned earlier, among variables in a Bayesian network, implying that relatively few belief values need be specified for each node, since they depend exclusively on its parents' belief values [Cha91]. The question still remains, however, how to obtain belief values initially, discussed next.

Ideally, prior belief values are determined by collecting empirical, historical or statistical data. This is possible in software projects that collect data on, for example, program bottlenecks and defect rates. Empirical data may also be available for development tasks, including requirements analysis, design, coding and testing. For example, empirical data regarding coverage adequacy of different testing criteria may be used to revisit the belief values in Table 1.

The ideal case, however, is seldom feasible. Instead, Bayesian belief values are usually elicited from domain experts who subjectively assess them. Domain experts include, among others, project managers, lead designers, senior programmers, and test researchers. Note that domain experts are used primarily to determine *prior* belief values; subsequent changes to belief values are caused by new evidence by way of Bayesian updating.

9 Conclusions and Future Work

The Maxim of Uncertainty in Software Engineering (MUSE) states that uncertainty is inherent and inevitable in software development processes and products. It is a general and abstract statement applicable to many facets of software engineering. We have chosen a probabilistic approach to uncertainty modeling, called Bayesian belief networks, and applied it to a software solution for an elevator control system. The Bayesian approach affords dynamic updating of beliefs during software development. We have designed and implemented rudimentary Bayesian-network capabilities for software systems. We have discussed some concerns and implications of the Bayesian approach for software engineering situations.

We believe that much more stands to be gained by explicit modeling of uncertainty in software engineering. In remaining paragraphs, we discuss additional uses and future research directions for uncertainty modeling.

9.1 Monitoring the testing process

An important question in software testing is "How much testing is enough?". This question may be addressed by explicit modeling of uncertainty, if sufficient testing is defined in terms of levels of confidence in select system entities, for example, its code modules. As testing progresses, confidence levels increase as long as test execution is successful. Testing is guided and monitored by continuous update and comparison of confidence levels against predefined thresholds. Testers are notified and may take appropriate action whenever thresholds are exceeded. This approach may be especially useful in safety-critical systems, where confidence requirements and constraints are often specified numerically.

9.2 Other software-engineering domains

In this paper, we have focused on software testing uncertainties, but we believe that uncertainty could and should also be modeled for other domains, including software reuse and re-engineering, requirements analysis and specification, software design and coding.

9.3 Other software qualities

In this paper, we have focused on validity, not correctness, as a software quality for which belief values are represented. We believe, however, that uncertainty should be modeled explicitly for many other software qualities, including correctness, reliability, fairness, safety, testability, maintainability, and efficiency. As mentioned earlier, qualities associated with entities must be consistent with causal relationships such that the resulting network is meaningful.

9.4 Other uncertainty modeling techniques

In this paper, we have used Bayesian networks to model uncertainty in software development. Viable alternatives to the Bayesian approach exist, however, including Certainty–Factor approaches, Dempster–Shafer approaches, fuzzy logic, and default and monotonic logic [Ste95]. Relative merits and pitfalls of these techniques should be studied and evaluated against the Bayesian approach in the context of software engineering situations.

9.5 Modeling uncertainty in software processes

The provision and update of belief values may be greatly enhanced in software process frameworks that include process measurement capabilities. Such capabilities constitute a rich source of information regarding the current state of various elements and support the collection of statistical and empirical data that may significantly improve the accuracy of prior belief value estimation.

We expect that by modeling software process uncertainties, one may achieve a more realistic representation of the process, enable automated belief revision by means of Bayesian updating, and support prediction and guidance of future development activities.

References

[Boe89] Barry W. Boehm, editor. *Software risk management.* IEEE Computer Society Press, Washington, D.C., 1989.

[Boe91] Barry W. Boehm. Software risk management: principles and practices. *IEEE Software*, 8(1):32–41, January 1991.

[Bro75] F. P. Brooks. *The Mythical Man-Month.* Addison-Wesley, Reading, MA, 1975.

[Bro87] Frederick P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[BW94] M. Blum and H. Wasserman. Program result-checking: A theory of testing meets a test of theory. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 382–391, Santa Fe, NM, 1994. IEEE Computer Science Press.

[Cha91] Eugene Charniak. Bayesian networks without tears. *AI Magazine*, pages 50–63, Winter 1991.

[Coo90] G. Cooper. Computational complexity of probabilistic inference using bayesian belief networks (research note). *Artificial Intelligence*, 42:393–405, 1990.

[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.

[Dav95a] Alan M. Davis. *201 Principles of Software Development.* McGraw Hill, New York, New York, 1995.

[Dav95b] Alan M. Davis. Tracing: A simple necessity neglected. *IEEE Software*, 12(5):6–7, September 1995.

[DBSB91] Premkumar T. Devanbu, Ronald J. Brachman, Peter J. Selfridge, and Bruce W. Ballard. LaSSIE: a knowledge-based software information system. *Communications of the ACM*, 34(5), May 1991.

[DKMS+92] Laura K. Dillon, George Kutty, P. Michael Melliar-Smith, Louise E. Moser, and Y.S. Ramakrishna. Graphical specifications for concurrent software systems. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 214–224, Melbourne, Australia, May 1992.

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

[Gai85] Jason Gait. A debugger for concurrent programs. *Software — Practice & Experience*, 15(6):539–554, June 1985.

[Gai86] Jason Gait. A probe effect in concurrent programs. *Software — Practice & Experience*, 16(3):225–233, March 1986.

[GJM91] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1991.

[GK94] H. Gall and R. Klösch. Managing uncertainty in an object recovery process. In *5th International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU'94)*, pages 1229–1235, Paris, France, July 1994.

[GKM95] H. Gall, R. Klösch, and R. Mittermeir. Object-oriented re-architecturing. In *5th European Software Engineering Conference (ESEC'95)*, pages 499–519, Barcelona, Spain, September 1995.

[Goe85] Amrit L. Goel. Software reliability models: Assumptions, limitations, and applicability. *IEEE Transactions on Software Engineering*, SE-11(12):1411–1423, 1985.

[GR95] Adele Goldberg and Kenneth S. Rubin. *Succeeding with objects : decision frameworks for project management*. Addison-Wesley, Reading, MA, 1995.

[Ham96] Dick Hamlet. Predicting dependability by testing. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA)*, pages 84–91, San Diego, CA, January 1996. ACM Press.

[HKBBS92] George T. Heineman, Gail E. Kaiser, Naser S. Barghuoti, and Israel Z. Ben-Shaul. Rule chaining in Marvel: dynamic binding of parameters. *IEEE Expert*, 7(6):26–33, December 1992.

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

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

[Kad92] R. Kadia. Issues encountered in building a flexible software development environment: Lessons learned from the Arcadia project. In *Proceedings of ACM SIGSOFT '92: Fifth Symposium on Software Development Environments*, pages 169–180, December 1992.

[Lit79] Bev Littlewood. How to measure software reliability and how not to. *IEEE Transactions on Reliability*, R-28(2):103–110, June 1979.

[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.

[MH89] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[MS91] Peiwei Mi and Walt Scacchi. Modeling articulation work in software engineering processes. In Mark Dowson, editor, *Proceedings of the First International Conference on the Software Process*, pages 188–201, 1991.

[MW96] Roland T. Mittermeir and Lydia G. Wurfl. Composing software from partially fitting components. In *IPMU'96*, pages 1121–1127, Granada, Spain, July 1996.

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

[PD91a] Rubén Prieto-Díaz. Domain analysis for reusability. In Rubén Prieto-Díaz and Guillermo Arango, editors, *Domain Analysis and Software Systems Modeling*, pages 63–69. IEEE Computer Society Press, 1991.

[PD91b] Rubén Prieto-Díaz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5):88–97, May 1991.

[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.

[PS92] B. Peuschel and W. Schäfer. Concepts and implementations of a rule-based process engine. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 262–279, Melbourne, Australia, May 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.

[Ric94] Debra J. Richardson. TAOS: Testing with analysis and oracle support. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)*, pages 138–153, Seattle, August 1994. ACM Press.

[Spi89] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, New York, 1989.

[Ste95] Mark Stefik. *Introduction to Knowledge Systems.* Morgan Kaufmann, San Francisco, CA, 1995.

[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.

[Ziv97] Hadar Ziv. *Bayesian-network Modeling of Software Testing and Maintenance Uncertainties.* PhD thesis, University of California, Irvine, 1997. Working Title, In Preparation.

[ZKR96] Hadar Ziv, René Klösch, and Debra J. Richardson. Software re-architecting in the presence of partial documentation. Technical Report UCI-TR-96-30, University of California, Irvine, August 1996.

[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.

[ZR97] Hadar Ziv and Debra J. Richardson. Software belief networks: Principles and practice. Technical report, University of California, Irvine, January 1997. To be submitted to SEKE'97.

[ZRK96] Hadar Ziv, Debra J. Richardson, and René Klösch. The uncertainty principle in software engineering. Technical Report UCI-TR-96-33, University of California, Irvine, August 1996.