The Uncertainty Principle in Software Engineering

Hadar Ziv **Debra J. Richardson**
Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425
USA
+1.714.824.4047
{ziv,djr}@ics.uci.edu

René Klösch
Department of Distributed Systems
Institute of Information Systems
Vienna University of Technology
Vienna, Austria
+43.1.58801
R.Kloesch@infosys.tuwien.ac.at

Abstract
This paper makes two contributions to software engineering research. First, we observe that uncertainty permeates software development but is rarely captured explicitly in software models. We remedy this situation by presenting the Uncertainty Principle in Software Engineering (UPSE), which states that uncertainty is inherent and inevitable in software development processes and products. We substantiate UPSE by providing examples of uncertainty in select software engineering domains. We present three common sources of uncertainty in software development, namely human participation, concurrency, and problem-domain uncertainties. We explore in detail uncertainty in software testing, including test planning, test enactment, error tracing, and quality estimation. Second, we present a technique for modeling uncertainty, called Bayesian belief networks, and justify its applicability to software systems. We apply the Bayesian approach to a simple network of software artifacts based on an elevator control system. We discuss results, implications and potential benefits of the Bayesian approach. The elevator system therefore serves as an example of applying UPSE to a particular software development situation. Finally we discuss additional aspects of modeling and managing uncertainty in software engineering in general.

Keywords
Software principles, software testing, uncertainty modeling, Bayesian networks

INTRODUCTION

Today's software engineer is expected to develop, maintain and comprehend software systems of unprecedented size and complexity. The complexity of software systems and their development processes is known to be intrinsic and essential [3]. Substantial efforts in software engineering research attempt to improve software quality and developer productivity in the presence of complexity. We contend that attempts to alleviate software complexity are often impeded by the uncertainty permeating virtually every aspect of software development. In subsequent sections, we provide supporting evidence for our claim. First, examples of uncertainty in select domains of software development are presented, followed by a discussion of three common sources of uncertainty in software engineering. We then present the Uncertainty Principle in Software Engineering (UPSE), followed by more detailed expositions of uncertainty in software testing, including test planning, test enactment, error tracing, and quality estimation. We describe a specific technique for modeling and management of uncertainty, known as Bayesian belief networks or simply Bayesian nets. We demonstrate the use of Bayesian nets for a simple network of software artifacts and relations based on an elevator control system. We conclude with a discussion of issues and concerns in uncertainty modeling, both specifically using Bayesian nets as well as in general.

We wish to note that uncertainty abounds in many aspects of software development, as well as in other engineering disciplines and in everyday situations (For uncertainty in everyday situations, see, for example, [33], pp. 460.). A detailed exposition of uncertainty in general is therefore beyond the scope of this paper. We hope that UPSE nevertheless identifies an opportunity for future investigations and provides a solid foundation for broader discussions of uncertainty modeling in software engineering. We encourage the reader to consider occurrences and influences of uncertainty in her own domains of interest and expertise.

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. For each domain, we include questions that arise frequently and indicate potential uncertainties. These questions often require answers of degree as opposed to binary yea or nay. Later, we show that these questions may be addressed by means of probability values.

Uncertainty in requirements analysis
Software requirements elicitation and analysis typically include learning about the problem and problem domain, understanding the needs of potential users, and understanding the constraints on the solution. Analysis and specification of software requirements inevitably introduces uncertainties, including: Who will be the real system users? What are users' needs and expectations? How well is the problem domain understood? How rigorously, accurately and sufficiently are domain understanding and user needs and expectations captured in the requirements specification document?

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?

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. 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 [14]? To what degree can original system documentation be used in the reverse engineering process [35]?

Uncertainty in software reuse
Reuse of software components (for example, classes, patterns and frameworks of object-oriented systems) introduces several uncertainties, including: How to specify the interface of a reusable component completely and sufficiently? What is one's confidence that an existing (i.e., available for reuse) component meets one's usage needs? Given a reusable component, how can it be tailored to existing project constraints and assumptions? Additional uncertainties have been identified in the composition of reusable components [25].

Sources of Uncertainty in Software Engineering
Uncertainty occurs in software engineering for different reasons and stems from multiple sources. Three example sources of uncertainty are described below.

Uncertainty in the problem domain
A software system typically includes one or more models of the "real world" domain in which it operates. The real world, however, is full of uncertainties, and therefore a system that models the real world inevitably reflects domain uncertainties. Domain uncertainties are often only implied or simply ignored in the system's domain model. This may lead to discrepancies between the real world and system assumptions and actions, which may in turn lead to risks and hazards. Also, in embedded systems, there are uncertainties with respect to external software, hardware, and mechanical components. Ignoring these uncertainties may be hazardous, possibly even fatal [1]. Uncertainty also exists in the natural and physical sciences. According to Heisenberg's uncertainty principle, for instance, the presence of an observer may affect scientific observations such that absolute confidence in observed results is not possible.

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, discussed next, occurs in concurrent-program debugging. The key difficulty in debugging concurrent programs is due to race conditions, which introduce uncertainty, because erroneous program behavior observed in one execution may not be evident in subsequent executions [24]. 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 is often referred to as the "probe effect" [12, 13] and is closely related to Heisenberg's uncertainty principle in that by attempting to observe program behavior, the probes may inadvertently affect the outcome of the observation. Consequently, this uncertainty has been referred to as "Heisenbugs" [24].

Human participation
The active role played by humans in virtually every stage of the software lifecycle inevitably introduces uncertainty and unpredictability into software development. That software development is still largely human-intensive may seem trivial, yet surprisingly few methods exist that explicitly model the inexact and uncertain nature of human involvement and its consequences for software processes and products. For example, rule-based formalisms for software-process modeling, such as Marvel/Oz [19] and Merlin [20], represent process steps and process decisions as rules, but do not accommodate explicit modeling of uncertainty in those rules.

[1] Leveson and Turner [22] report on the Therac incidents, where software uncertainties were deemed impossible and therefore were not considered by the manufacturer of a medical radiation device.

Principles of Software Engineering

In 1968, the NATO conference held in Garmisch, Germany [26] endorsed the claim that software construction is similar to other engineering tasks and that software development must therefore "be practiced like an engineering discipline." Engineers in classical engineering disciplines are equipped with processes, methodologies, standards, and tools that have been evolved, tested, and proven successful. The use of standard procedures, materials, and building blocks limits the degrees of freedom and allows for engineering projects to proceed in predictable, controllable, and manageable fashion.

At the foundation of classical engineering disciplines one often finds a small set of underlying principles and laws of nature that govern the behavior of systems and guide their development. In the physical sciences, for instance, one finds Newton's laws of gravity, Einstein's theory of relativity, Kepler's laws of planetary motion, Heisenberg's uncertainty principle, and the laws of thermodynamics. Laws and principles of the physical world are usually discovered, not invented, by observing physical systems. Such principles are confirmed and substantiated by scientific experiments that are controllable and repeatable and whose results are highly predictable.

In contrast, software systems appear unconstrained by any laws or principles. It has long been recognized, however, that software engineering would do well by a standard set of procedures, guidelines and principles. A recent book by Davis [8] documents 201 such principles. Ghezzi et al [15] identify seven principles at the heart of all software development activities. Brooks contributed two key principles to software engineering knowledge, namely "the mythical man-month" [2] and "no sliver bullet" [3].

Software engineering principles should capture the nature and behavior of software systems and guide their development. Such principles would help in restricting degrees of freedom in software development and achieving degrees of predictability and repeatability similar to those of classical engineering disciplines. We observe that in order for a principle of software engineering to exhibit relevancy and applicability similar to other engineering principles, it should be defined and presented generally and abstractly, be applicable and instantiable in practice to specific software development situations, and be observed and substantiated repeatably and predictably.

One can confirm that software-engineering principles, such as those recorded by Davis, Ghezzi, Brooks, and others, indeed meet the three "principle criteria" defined above. In this paper, we define and justify UPSE and contend that it also meets the three principle criteria above. We now proceed to define UPSE.

THE UNCERTAINTY PRINCIPLE IN SOFTWARE ENGINEERING

Software development is a complex human enterprise carried out in problem domains and under circumstances that are often uncertain, vague, or otherwise incomplete. Development must progress, however, in the presence of those uncertainties. The Uncertainty Principle in Software Engineering (UPSE) is therefore stated as follows:

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

UPSE is, like other principles of software engineering, a general and abstract statement about the nature of software development and is applicable throughout the development lifecycle. The principle should still, however, be applied judiciously and appropriately. The next section describes key issues and concerns that need to be addressed when applying UPSE.

Applying UPSE

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 UPSE by explicitly modeling one or more uncertainties, taking into account the issues discussed below.

What is the goal of the modeling activity?

A model of a software process or product is an abstraction that ignores some detail. Software models are developed for different reasons and to meet different goals and, consequently, may include different uncertainties. If, for example, the goal of software modeling is to represent key system abstractions or artifact architecture, then uncertainties regarding conceptual and architectural decisions may need to be represented explicitly. Alternatively, if a software process is modeled to facilitate project planning and prediction, then uncertainties regarding schedule and budget estimates, progress monitoring, and project risks may be modeled explicitly.

When is uncertainty modeling relevant?

Despite the generality of UPSE, uncertainty-modeling is not necessarily meaningful or equally applicable to all aspects of a software-engineering effort. Consider, for example, a requirements-change scenario where a new feature is requested for an existing software system. The challenging task of accommodating the new requirement often necessitates substantial changes to system architecture and implementation, and therefore leads to uncertainties. On the other hand, a simpler, automatable task such as creating a new release version or updating system configuration information is less likely to introduce uncertainty [2].

What notation, formalism, or approach is used for modeling uncertainty?
Notations and techniques for modeling uncertainty, vagueness, and fuzziness, have been studied extensively in domains of artificial intelligence [33, 18]. Different approaches for modeling uncertainty have been advocated based on different mathematical models as well as different assumptions about the sources and nature of uncertainty. A detailed exposition of current approaches to uncertainty modeling is beyond the scope of this paper. Instead, we focus on a particular uncertainty modeling technique, called Bayesian belief networks.

UNCERTAINTY IN SOFTWARE TESTING

Software testing has been described as "the search for discrepancies between what the software can do versus what the user or the computing environment wants it to do" [16]. We consider software testing broadly to include test planning, test enactment, error tracing, and quality estimation. We identify uncertainties associated with each activity below.

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, design representations, source code modules, and the relationships among them. These software artifacts are produced by requirements analysis, architectural design, and coding processes, respectively. Following UPSE, uncertainty permeates those development processes and, consequently, the resulting software artifacts. Plans to test them, therefore, will carry these 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. This again introduces uncertainties.

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.

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 test results against stated specifications. Test oracles can be classified into five categories [31], listed in decreasing order of uncertainty (or, alternatively, increasing order of confidence), as follows: human oracles, input/output oracles, regression test suites, validation test suites, and specification-based oracles. Specification-based oracles instill the highest confidence, but still include uncertainty that stems from discrepancies between the specification and customer's informal needs and expectations.

We have modeled uncertainties in test oracles for an extended system test scenario, but space does not permit its inclusion in this paper. Instead, we provide two smaller and simpler examples of modeling uncertainty in software testing. The first example, described next, is in the domain of path selection criteria.

Example: Path Selection Testing Criteria

Here, we add a measure of uncertainty to a previous

[2] Note, however, that automatable operations that do not require human intervention are not necessarily free of uncertainties.

result in comparison of data flow path selection testing criteria. In [5], 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 may be recast in terms of uncertainty or degree of confidence, as follows: if criterion A subsumes criterion B, then one has more confidence in the defect-detection ability of A than that of B[3]. Confidence in the defect-detection ability of a given testing criterion may be quantified by means of a probability "belief" value between 0 and 1. This is illustrated in Table 1, which shows a plausible assignment of probabilistic confidence values for a dozen path selection criteria from [5]. Table 1 raises some important questions, however, discussed next.

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

Why are confidence values relatively low?
Low confidence values imply that even a "strong" path selection criterion does not incur high levels of confidence. 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. Therefore, low confidence values reflect the criteria's inability to guarantee defect detection.

What are the constraints, if any, on the assignment of confidence values?
The only constraint on assigning confidence values is that if A subsumes B in the subsumption hierarchy of [5], then A's confidence value should be equal to or higher than B's. Thus, there are infinitely many possible assignments of confidence values that preserve the partial subsumption order of path selection criteria.

[3] As discussed in [5], even if A subsumes B, it is still uncertain whether A is in fact better at detecting defects, since demonstrating the latter would require that empirical data be collected to substantiate the graph theoretic proofs of subsumption.

How are confidence values determined?
Confidence values, such as shown in Table 1, are often determined by consultation with domain experts. Other techniques exist, however, for establishing confidence values, including: values computed using software reliability or cost-estimation models; values obtained from empirical, statistical, or historical data; or else values acquired dynamically during software-process execution. Some techniques and their implications are discussed further in subsequent sections.

How are confidence values used?
Confidence values may be useful, for example, for choosing the most appropriate testing criterion given project requirements and constraints. A safety-critical system, for instance, may require that only testing criteria with confidence levels in the ultrahigh region be used. In contrast, a commercial software product may may weigh the cost and duration of testing against time-to-market constraints. We propose that, in both cases, probabilistic measures of confidence, for example, in the defect-detection abilities of testing criteria, be employed in the decision-making process.

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 by means of quality estimation. We consider reliability testing and reliability growth modeling to be among the most mature techniques for software quality assessment [23] 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 [16]. 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 [16].

Hamlet [17] and Littlewood [23] extend existing reliability theory be defining "software dependability" as a statistical measure of software quality. Hamlet incorporates Blum's idea of self-checking/self-correcting programs [1] into reliability such that the dependability of a program P at input X is defined as the confidence probability that P is correct (with respect to its specification) at X.

Software reliability models not only demonstrate that uncertainty may be measured and represented explicitly but they can also be used to estimate future software quality. Prediction of future reliability assumes that software systems are used with statistical regularity. This assumption, however, introduces uncertainty, since future users may exhibit vastly different usage patterns. We conclude that probabilistic measures of software reliability can be used to provide initial estimates of confidence levels in software artifacts and relations. This is discussed in more detail in subsequent sections.

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" [10], requires that software artifacts are interrelated among themselves as well as to informal customer requirements.

Software traceability is therefore the creation, management, and maintenance of relations from one software entity to other entities [9]. Software development environments, including, among others, Marvel/Oz [19], Merlin [20], and Arcadia [21], support software traceability by means of tool integration, object management systems, and hypertext capabilities. 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?" [36, 34]. This difficulty is akin to the hypertext-navigation problem known as "lost in hyperspace" [28].

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.

MODELING UNCERTAINTY

We suggest that uncertainties associated with one or more properties of software artifacts be modeled and maintained explicitly. A network of software artifacts, annotated with uncertainty values, can then, for example, be navigated more effectively by guiding the software engineer to artifacts that are more likely to exhibit a particular property. We now describe the Bayesian approach to uncertainty modeling.

Bayesian Belief Networks

Bayesian belief networks have been used in artificial intelligence research to provide a framework for reasoning under uncertainty [29, 27]. Bayesian networks have been used extensively in a wide range of applications [18]. Specifically, the Bayesian approach has been applied successfully to large text and hypertext search databases in the domain of information retrieval [11, 7] and to validation of ultrahigh dependability for safety-critical systems [23].

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 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 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 [29], 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. This updating scheme supports distributed implementation, since each node can execute in a separate execution thread and be updated by way of message passing.

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 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.

Bayesian updating of an arbitrary network (i.e., where cycles may exist in the underlying undirected graph) is known to be NP–hard [6], but if the network is tree-structured [5], Pearl's updating algorithm is quadratic in the number of values per node and linear in the number of children per parent. For a more comprehensive description of Pearl's updating algorithm, see [29, 27].

Why Bayesian Networks?

We identify compelling reasons for using Bayesian networks for modeling uncertainty in software engineering. First, it is a mechanism to apply UPSE 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 imply that other uncertainty-modeling techniques should not be considered. Rather additional investigation of other approaches is required in order to study their possible uses and compare their relative merits versus the Bayesian approach.

THE ELEVATOR SYSTEM EXAMPLE

As part of a large effort to demonstrate integration capabilities of the Arcadia research project [21], we have developed a complete software solution for an elevator control system. The elevator system is a classic problem that has been used to demonstrate software engineering techniques in general and specifically in the area of formal specification languages [32, 31]. The elevator system is required to control n elevators in a building with m floors. The problem concerns the logic required to move elevators between floors according to specified functional requirements as well as safety, liveness, and fairness constraints.

Software artifacts in our elevator system solution include a functional decomposition of requirements, developed using REBUS; formal requirements specifications, including model-based specifications in Z and interval logic specifications using both RTIL and the GIL toolset; object-oriented design diagrams, using Software Through Pictures' OOSD/Ada notation; code modules implemented in Ada; and test suites, test criteria, and test oracles, developed using TAOS [30].

Software artifacts in the elevator system are interrelated by means of artifact relationships, as follows: Ada code units are related to OOSD design specification elements; design specifications are related to requirements specifications; requirements specifications are related to test suites and test oracles that are used to ensure that the system meets specified requirements; test suites are related to code units that are to be tested against the requirements; and test criteria, used to determine whether the code is adequately tested, are related to code units and test suites.

We applied the Bayesian approach to the elevator system solution. Software artifacts and relations were assigned probability values that were determined by con-

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

[5] In this paper, we limit our discussion to tree-structured software networks. Bayesian updating algorithms for general DAGs exist, however, and are polynomial in time and space.

sultation with a domain expert. Though we have assigned belief values and carried out Bayesian updating for the entire elevator system, space does not allow for the complete example to be shown. Instead, for clarity and brevity, we demonstrate the Bayesian approach for a partial unit test scenario that is modeled by a subnetwork of only four elevator-system artifacts. The complete elevator example can be found in [37].

The Unit Test Scenario

In the unit test scenario, a software entity is considered valid if it is traceable to original customer requirements and meets customer needs and expectations (cf. [15]). Note that absolute confidence in an entity's validity is hard to achieve in practice. Instead, we associate a probabilistic belief value with the statement "this entity is valid," and assign those belief values to system entities accordingly.

In the unit test scenario, design node D represents an OOSD design specification element, for example, *Elevator_Controller_Interface_Spec*. A probability value is associated with D, representing prior belief that D is valid. Similarly, module M represents an Ada code unit, for example, *Elevator_Controller_Package*, and is assigned a probability value representing prior belief that M is valid. Since M implements D, there exists a causal relationship between M and D, indicated by a directed edge in the network of Figure 1.

In addition, test nodes $T1$ and $T2$ represent two test suites, corresponding to two different test selection criteria, for example, *All–Edges* and *All–Uses*. Test suites are executed against code units in the system's implementation and may succeed or fail. Test suite execution is successful 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 is considered invalid if a single defect is detected [6], i.e., if execution of any related test suite fails, which, correspondingly, sets its belief value to zero. Note, however, that successful test suite execution does not set the corresponding module's belief value to one, since it does not instill complete confidence. Rather, confidence that M is valid merely increases with each successful test suite execution. This is confirmed by the results of Bayesian updating in the unit test scenario, reported below.

The software network of Figure 1 provides a computational framework for updating beliefs in the validity of entities. In particular, success or failure of test suite execution constitutes new evidence that is then propagated throughout the network to revise previous beliefs. The initial state of the network, described next, includes prior beliefs in network nodes, as determined by consultation with a domain expert.

[6] Alternate definitions of validity are discussed later.

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 follows:

$$Bel(D) = \begin{pmatrix} Bel(D = valid) \\ Bel(D = invalid) \end{pmatrix} = \begin{pmatrix} .7 \\ .3 \end{pmatrix}$$

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 values are initially set to the same values as $Bel(D)$. Similarly, since no propagation has occurred yet, D's λ values are all set to 1.

A directed edge from D to M indicated 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). These probabilities are determined by a domain expert as follows: if D is valid, then M is also valid with .6 probability. The probability that M is invalid is, or course, .4. If D is invalid, however, then M is valid with only .1 probability and invalid with .9 probability. These probabilities are recorded in the edge matrix between D and M, as shown in Figure 1.

Next, we compute our belief that M is valid by way of downward propagation. This is accomplished by computing a π vector for M by multiplying D's π vector (the downward message) by the transpose of the edge matrix between D and M. The resulting π values are assigned to *Bel(M)* and indicate initial belief of 45% in M's validity. This is shown below and in Figure 1.

$$Bel(M) = \begin{pmatrix} .6 & .4 \\ .1 & .9 \end{pmatrix}^T \begin{pmatrix} .7 \\ .3 \end{pmatrix} = \begin{pmatrix} .45 \\ .55 \end{pmatrix}$$

Test suite $T1$ represents *All–Edges*, a relatively weak testing criterion in the subsumption hierarchy of [5]). Table 1 associates a confidence level of .15 with the defect-detection ability of *All–Edges*. We therefore determine the following probabilities for the edge matrix between M and $T1$: if M is invalid, then $T1$ succeeds (i.e., executes successfully) with .85 probability. Correspondingly, $T1$ fails with .15 probability. If M is valid, then $T1$ always succeeds. Similarly, test suite $T2$ represents *All–Uses*, a stronger testing criterion. Table 1 associates a confidence level of .45 with the defect-detection ability of *All–Uses*. This again determines the corresponding edge-matrix probabilities to be .45 and .55, respectively. The resulting edge matrices are 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 $T2$.

Network State After Executing $T1$
Figure 2 illustrates the effects on the network of successful execution of test suite $T1$. Bayesian updating proceeds by means of sending λ and π messages up and down the network, as follows: $T1$'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; 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, yielding a revised belief value of .49 that M is valid. M then sends a λ message to its parent D, which is used to revise D's λ and *Belief* vectors, as before. The revised belief that D is valid is .72. Finally, M also sends a π message to $T2$, where the π values are identical to M's new belief values. $T2$ then recomputes its own π and belief vectors.

Network State After Executing $T2$
Next we consider the effects on the network of executing the stronger test suite $T2$ (*All-Uses*). Whether $T2$ succeeds or fails, belief values in the network are updated by means of propagation and re-computation of λ and π values. If $T2$ were to fail, a defect has been detected and M is recognized as invalid. Specifically, $T2$'s λ vector is set to $\binom{0}{1}$ upon failure, and, after multiplication by the edge matrix, updates M's λ and belief vectors to also be $\binom{0}{1}$. Additional upward propagation from M to D results in a decrease in our belief in the validity of D. But, if $T2$ succeeds, then M's belief value increases to 96.5%, and our belief that D is valid increases to 97%.

DISCUSSION

The application of a Bayesian or other probabilistic approach to software systems raises some issues and concerns. Among those we discuss issues deemed most pertinent to this paper (in no particular order).

How are belief values interpreted?

In most applications of Bayesian networks (cf. [18]), belief values are associated with observable phenomena, described using binary True/False statements. When modeling everyday situations, for example, the prior belief value of the statement "It is sunny" may be determined to be .9, while the belief value of the statement "The dog is barking" may be .55 [4]. Each statement can therefore be viewed as an observation 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 for the design node and code unit is quality, 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. In the unit test scenario, for example, test suites are used to test the validity of code units, and therefore the observed quality is validity.

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.

Assume, for example, that a Bayesian value represents belief that a source code unit is "bug free" or otherwise correct 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 (as in the unit test scenario above). 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.

Assume a different scenario where a complex software system includes many modules and is developed under stringent schedule constraints. Here, it might be acceptable for code units to contain known defects given certain project considerations, including, for example, "How many defects were detected in the module?", "What kind of defects were detected?", "How costly is defect elimination during development?", and "How costly would this defect be if it caused operational failure?". 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.

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 [4]. 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 a domain expert who subjectively assesses them. Domain experts include, for example, project managers, lead programmers, senior designers, test researchers for test-strategy effectiveness, and so on. Note that domain experts are used primarily to determine *prior* belief values; subsequent changes to belief values in the network are caused by new evidence by way of Bayesian updating.

CONCLUSIONS AND FUTURE WORK

The Uncertainty Principle in Software Engineering (UPSE) states that uncertainty is inherent and inevitable in software development processes and products. UPSE is a general and abstract statement about the nature of software development. To demonstrate UPSE, we have chosen a probabilistic Bayesian approach to uncertainty modeling and applied it to a simple software network based on an elevator system. The Bayesian approach affords dynamic updating of beliefs during software development. 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 this paper, we have merely posited UPSE and demonstrated its applicability, using the Bayesian approach as a point example. In the remaining paragraphs, we discuss additional uses and future research directions for uncertainty modeling.

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.

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.

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.

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 [33]. Relative merits and pitfalls of these techniques should be studied and evaluated against the Bayesian approach in the context of software engineering situations.

Modeling uncertainty in software process
In this paper, we have demonstrated that uncertainty can be modeled for both process (i.e., testing strategies) as well as product (i.e., artifact networks) aspects of software development, with an emphasis on modeling uncertainty in software products. With respect to modeling uncertainty in software processes, we believe that software-process modeling formalisms must be augmented to include uncertainty values; that an environment for supporting definition and execution of process models should include capabilities for representation and interpretation of belief values and should allow for Bayesian updating of those values; and that Bayesian updating procedures must be carried out during process execution, such that belief values and confidence levels are continuously updated as new evidence arrives.

The provision and update of belief values may be greatly enhanced in 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

[1] 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*, Santa Fe, NM, 1994. IEEE Computer Science Press.

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

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

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

[5] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, SE-15(11), November 1989.

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

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

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

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

[10] P. T. Devanbu, R. J. Brachman, P. J. Selfridge, and B. W. Ballard. LaSSIE: a knowledge-based software information system. *Communications of the ACM*, 34(5), May 1991.

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

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

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

[14] 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.

[15] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1991.

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

[17] D. 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.

[18] D. Heckerman, A. Mamdani, and M. P. Wellman. Real-world applications of bayesian networks. *Communications of the ACM*, 38(3), March 1995.

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

[20] G. Junkermann, B. Peuschel, W. Schäfer, and S. Wolf. *MERLIN: Supporting Cooperation in Software Development Through a Knowledge-Based Environment*, chapter 5, pages 103–129. Wiley & Sons, England, 1994.

[21] 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.

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

[23] B. Littlewood and L. Strigini. Validation of ultrahigh dependability for software-based systems. *Communications of the ACM*, 36(11), November 1993.

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

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

[26] P. Naur, B. Randell, and J. N. Buxton, editors. *Software Engineering: Concepts and Techniques: Proceedings of the NATO Conferences*. Petrocelli-Charter, New York, New York, 1976.

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

[28] J. Nielsen. *Multimedia and Hypertext : the Internet and beyond*. AP Professional, Boston, MA, 1995.

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

[30] D. 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.

[31] D. J. Richardson, S. L. Aha, and T. O. 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.

[32] S. R. Schach. *Classical and Object-Oriented Software Engineering*. Irwin, Chicago, Illinois, 1996.

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

[34] D. Steinberg and H. Ziv. Software Visualization and Yosemite National Park. In *Proceedings of the Twenty-Fifth Annual Hawaii International Conference on System Sciences*, January 1992.

[35] H. Ziv, R. Klösch, and D. J. Richardson. Software re-architecting in the presence of partial documentation. Technical Report UCI-TR-96-30, University of California, Irvine, August 1996.

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

[37] H. Ziv, D. J. Richardson, and R. Klösch. The uncertainty principle in software engineering. Technical Report UCI-TR-96-33, University of California, Irvine, August 1996.

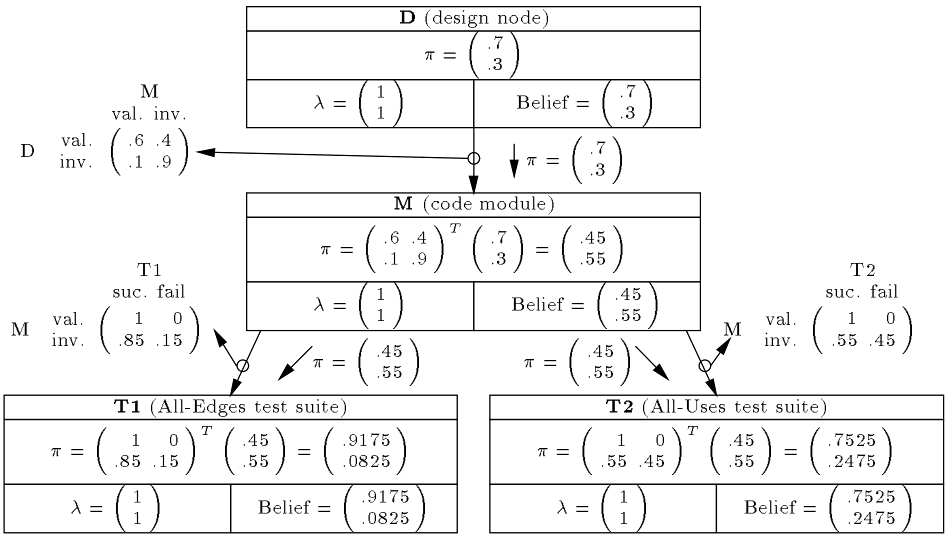

Figure 1: Initial Belief Network for Unit Test Scenario

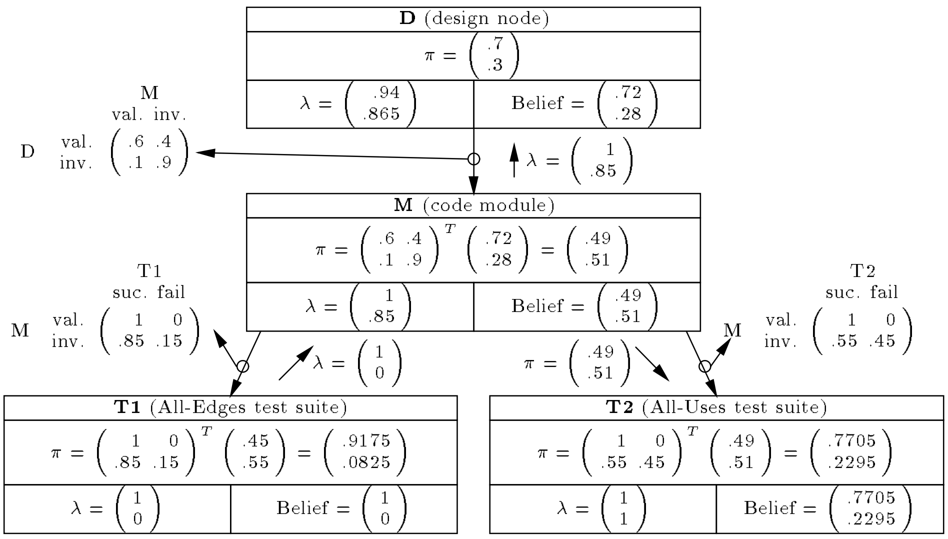

Figure 2: Revised Belief Network After Execution of $T1$