Using Bayesian Belief Networks in Assessing Software Architectures

Jilles van Gurp & Jan Bosch
[Jilles.van.Gurp|Jan.Bosch]@ipd.hk-r.se
University of Karlskrona/Ronneby
Department of Software Engineering and Computer Science
Soft Center, S-372 25 Ronneby
http://www.ipd.hk-r.se/[jvg|bosch]

Abstract. *Quantitative techniques have traditionally been used to assess software architectures. We have found that early in the development process there is often insufficient quantitative information to perform such assessments. So far the only way to make assessments about an architecture, is to use qualitative assessment techniques like for instance a peer review. The problem with this type of assessment techniques is that they depend on the knowledge of the expert designers who use them. In this paper we introduce a technique that automates making qualitative assessments of software architectures.*

1 Introduction

Traditionally the development of software is organized into different phases. The phases usually occur in a linear fashion (the waterfall model). Often the phases of this model are repeated in an iterative fashion. This is especially true for Object Oriented systems. During the requirement specification phase both functional and non-functional requirements are collected. Based on these requirements the design is conceived. After the design phase, the design is implemented. Then the implementation is tested. This test is also the final verification that the product meets the requirements. After that the product is deployed. After the product deployment the product has to be maintained. Maintenance can be bug fixing (corrective maintenance) but can also involve optimizations (perfective maintenance) and adaptations (adaptive maintenance) [18].

At any phase in the process, the process can shift back to an earlier phase. If, for instance, during testing a design flaw is discovered, the design phase and consequently also the implementation phase need to be repeated. It is also possible that developers discover during implementation that meeting a certain requirement is not feasible (for instance because it is too expensive) in which case requirements need to be renegotiated. This type of setback in the software development process can be costly, especially if radical changes in the earlier phases (triggering even more radical changes in consequent phases) are needed. We have found that non-functional requirements or quality requirements often cause these type of setbacks. The reason for this is that testing whether the product meets the quality requirements does not occur until the testing phase.

To assess whether a system meets certain quality requirements, there are several assessment techniques. Most of these techniques are quantitative in nature. I.e. they measure properties of the system. Quantitative assessment techniques are not very well suited for use early in the development process, however. The reason for this is that incomplete products like design documents and requirement specifications provide not enough quantifiable information to perform the assessments. Instead developers resort to qualitative assessment techniques. A frequently used technique is, for instance, the peer review where design and or requirement specification documents are reviewed by a group of experts. Though these techniques are very useful in finding the weak spots in a system, many flaws go unnoticed until the system is fully implemented. Fixing the architecture in this stage can be very expensive because by then the system has become complicated.

Qualitative assessment techniques, like the peer review, rely on qualitative knowledge. This knowledge resides mostly in the heads of developers and may consist of solutions for certain types of problems (patterns [2][6]), statistical knowledge (60% of the total system cost is spent on maintenance), likely causes for certain types of problems (“our choice for the broker architecture explains weak performance“), aesthetics (“this architecture may work but it just doesn’t feel right“), etc. A problem is that this type of knowledge is inexplicit and very hard to document. Consequently, qualitative knowledge is highly fragmented and largely undocumented in most organizations. There are only a handful known ways to handle qualitative knowledge:

- Assign experienced designers to a project. Experienced designers have a lot of knowledge about how to engineer systems. Experienced designers are scarce, though, and when an experienced designer resigns from the organization he was working for, his knowledge will be lost for the organization.
- Knowledge engineering. Here organizations try to capture the knowledge they have in documents. This method is especially popular in large organizations since they have to deal with the problem of getting the right information in the right spot in the organization. A major obstacle is that it is very hard to capture qualitative knowledge as discussed above.
- Artificial Intelligence (AI). In this approach qualitative knowledge is used to built intelligent tools that can assist personnel in doing their jobs. Generally, such tools can't replace experts but they may help to do their work faster. Because of this less experts are needed.

In this paper we present a way for representing and using qualitative knowledge in the development process. The technique we use for representing qualitative knowledge, Bayesian Belief Networks, originates from the AI community. We have found that this technique is really good at modeling and manipulating the type of knowledge described above. Bayesian Belief Networks are currently used in many organizations. Examples of such organizations are NASA, HP, Boeing, Siemens [8]. BBNs are also applied in Microsoft's Office suite where they are used to power the infamous paperclip [13].

We created a Bayesian Belief Network, called SAABNet (Software Architecture Assessment Belief Network), that enables us to feed information about the characteristics of an architecture to SAABNet. Based on this information, the system is able to give feedback about other system characteristics. The SAABNet BBN consists of variables that represent abstract quality variables such as can be found in MC Call's quality factor model [12] (i.e. maintainability, flexibility, etc.) but also less abstract variables from the domain of software architectures like for instance inheritance depth and programming language. The variables are organized in such a way that abstract variables decompose into less abstract variables.

In section 2 we give a short introduction to BBNs. In section 3 we introduce a belief network for processing qualitative information. In section 4 we validate this network using a few cases. Related work is presented in section 5 and we conclude our paper in section 6.

2 An introduction to Bayesian Belief Networks

A BBN is a directed acyclic graph. The nodes in the graph represent probability variables and the arrows represent dependencies (not causal relations!). If two nodes are not directly connected by an arrow, this means they are independent given the nodes in between.

Each node can contain a number of states. A conditional probability is associated with each of these states for each combination of states of their direct predecessors (see figure 1 for an example). By using a smart algorithm, the conditional chances for all of the variables in the network can be calculated, something that would take exponential amounts of processing power using conventional math (it's a NP complete problem). A BBN can be used by entering evidence (i.e. setting probabilities of variables to a certain value). The chances for the states of the other variables are then recalculated. How this is done is beyond the scope of this paper, however (for an introduction to BBNs we refer to [16]). Instead we will focus on how to build and use these networks for qualitative assessment of software architectures. A BBN consists of both a qualitative and a quantitative specification. The qualitative specification is the graph of all the nodes. The quantitative specification is the collection of all conditional chances associated with the states in each node.

In figure 1 an example of a very small BBN is given that we will use to explain the concepts of a BBN. In this figure there are four variables. Each variable in our example has only two states (true and false). To the left, a qualitative version of the network is drawn, to the right all the probabilities that are needed for the quantitative part of the network are given. Each node in a BBN has to specify the conditional probabilities for each of its states given a combination of its direct parent states. In our example A and B don't have any parents so there only are two probabilities. C has both A and B as a parent and thus has to specify 8 different probabilities. The figure only lists four but the remaining four (C=false) can be derived from these (since the probabilities for both states have to add up to 1). The same goes for variable D. Note that variable D does not depend on variable A. Variable D is called conditionally independent of variable A given variable C. Because of this concept it is possible to create very large net-

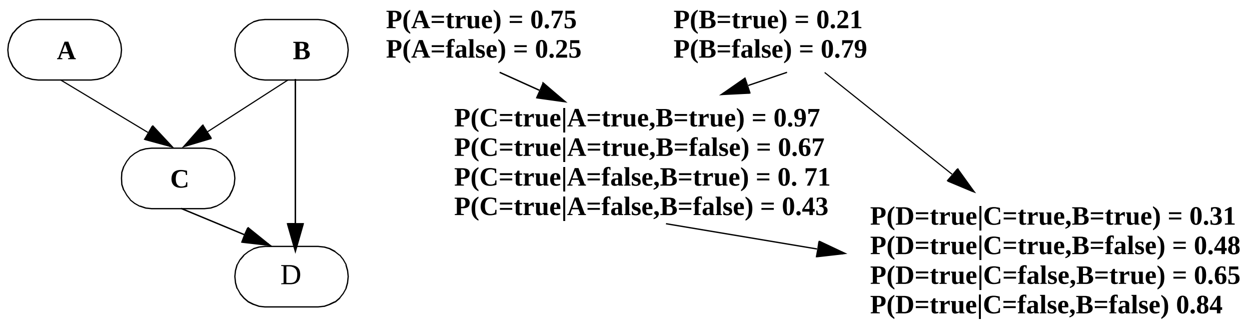
A
B
C
D
P(A=true) = 0.75
P(A=false) = 0.25
P(B=true) = 0.21
P(B=false) = 0.79
P(C=true|A=true,B=true) = 0.97
P(C=true|A=true,B=false) = 0.67
P(C=true|A=false,B=true) = 0. 71
P(C=true|A=false,B=false) = 0.43
P(D=true|C=true,B=true) = 0.31
P(D=true|C=true,B=false) = 0.48
P(D=true|C=false,B=true) = 0.65
P(D=true|C=false,B=false) 0.84

FIGURE 1. A small BBN

works without the calculations getting too complex. With modern computers it is possible to calculate networks with hundreds of nodes.

With this network and the algorithms that are used in BBNs it is possible to calculate probabilities like P(D=false|C=false,B=true) with variable A unknown or P(A=true|B=true) with variable C and D unknown. This can be done by entering the probability information behind the bar into the network. The BBN algorithm will than calculate the probabilities. Feeding probability information to the network is usually referred to as entering evidence. In the example above, entering the evidence P(C=false)=1 and P(B=true)=1 enables the algorithm to calculate the probabilities for the states in all the other variables given these two variables. This is called propagating evidence through the network.

A BBN is typically used by entering information that is known. I.e. the chances for states of variables are set to a value. The value can be 1 if the knowledge is 100% certain but may also be set to other values. The BBN then uses the conditional probabilities in the quantitative specification to recalculate all the probabilities of other variables' states. These recalculated probabilities form the output of the network. The more evidence is fed to the network, the more accurate the output.

2.1 What type of knowledge can be put in a BBN?

The nature of human knowledge is that it is unstructured, incomplete and fragmented. These properties make that it is very hard to make a structured, complete and unfragmented mathematical model of this knowledge. The strength of BBNs is that it enables us to reason with uncertain and incomplete knowledge. The problem of fragmentation still exists for this way of modeling knowledge, though.

To build a BBN, knowledge from many different sources has to be collected. In our case the knowledge resides in the heads of developers but there may also be some knowledge in the form of books and documentation. Examples of sources for knowledge are:

- Patterns. The pattern community provides us with a rich source of solutions for certain problems. Part of a pattern is a context description where the author of a pattern describes the context in which a certain problem can occur and what solutions are applicable. This part of a pattern is the most useful in modeling a BBN because this matches the paradigm of dependencies between variables.
- Experiences. Experienced designers can indicate whether certain aspects in a software architecture depend on each other or not, based on their experience.
- Statistics. These can be used to reveal or confirm dependencies between variables.

2.2 Construction of BBNs

Constructing a BBN generally involves the following steps:

- Identify relevant variables in the domain
- Define/identify the probabilistic dependencies and conditional independencies between the variables. (this should lead to a qualitative specification of the BBN)
- Assess the conditional probabilities (this should lead to a quantitative specification of the BBN)
- Test the network to verify that the output of the network is correct.

We have found that the last two steps need to be iterated many times and sometimes enhancements in the qualitative specification are needed.

Basically the only way to establish whether a BBN is reliable is to do casestudies. Doing such case studies means feeding evidence of a number of selected cases to the network and verifying whether the output of the network corresponds with the data available from the case studies. The network can be relied upon to deliver mathematical correct probabilities given correct qualitative and quantitative specifications of the BBN. If a BBN doesn't give correct output, that may be an indication that the probabilistic information in the network is wrong or that there is something wrong with the qualitative specification of the network.

Problems with the qualitative specification may be missing variables (over simplification) or incorrect dependency relations between variables (missing arrows or to many arrows). Problems with the quantitative specification are caused by incorrect conditional probabilities. Estimating probabilities is something that human beings are not good at so it is not unlikely that the quantitative specification has errors in it. Most of these errors only manifest them in very specific situation, however. Therefore a network has to be tested very thoroughly to make sure the output of it correct.

2.3 Interpreting the output of a BBN

It is important to realize that any model is a simplification of reality. Therefore, the output of a BBN is also a simplification of reality. When we designed our SAABNet network, we aimed to get realistic output. I.e. output that stresses good points and bad points of the architecture.

The output of a BBN consists of probabilities for each state in each variable. The idea is that a user enters chances for some of the variables (for instance P(implementation_language=Java)=1.0). This information is then used together with the quantitative specification of the network to calculate all the other chances. Since also chances other than 1.0 can be entered, the user is able to enter information that is uncertain.

Though the output of the network in it self is quantitative, the user can use this output to make qualitative statements about the architecture (“if we choose the broker architecture there is a risk that the system will have poor performance and higher complexity“) based on the quantitative output.

Sometimes the output of a BBN contradicts with what one expects from the given input. Contradicting output always has a cause:

- The model is wrong. This means that either there’s something wrong with the qualitative model. I.e. some variables are not taken into account that really are of importance. Or the quantitative model is wrong. I.e. the influence of some variables is overrated/underrated. Either error can be fixed by adapting the network.
- There is not enough input for the model to come up with a reliable estimation. In this case more evidence needs to be added.
- The entered evidence does not match reality. In this case the BBN cannot be expected to come up with reasonable estimates.
- The perceived definition of a term does not match the BBNs definition. This is a matter of understanding our BBN. Maybe the BBN needs to be changed to reflect the perceived definition or maybe some additional documentation may resolve the misunderstanding.
- The user is wrong. If none of the above is the cause of the contradiction, the user is wrong. Maybe the user has a different understanding of the variables in the network. Maybe the user knows something that was not entered into the network. Maybe the user assumes a causal connection that is not there.

In many cases the BBN will give neutral output. I.e. the probabilities for each state in a certain variable are more or less equal. Possible causes for this are:

- There is not enough information available to favor any of the states of this variable.
- The variable has no incoming arrows. This variable might be intended as an input variable. Especially when the number of outgoing arrows is low, it is unlikely that this variable is affected by the rest of the model very much.

If all of the above things are solved, the BBN may give high probabilities for some variable states. If this happens (assuming it does not conflict with reality), the network has provided useful output. It may confirm what had already been suspected in which case proper argumentation can be found for this suspicion by examining the rest of the network. It may also provide new information in which case it is certainly recommended to find out why the BBN gives a high probability.

3 SAABNet

Based on a number of cases we have created a BBN for assessing software architectures called SAABNet (Software Architecture Assessment Belief Network). The aim of SAABNet is help developers perform qualitative assessments. Its primary aim is to support the architecture design process. Consequently, it does not support later phases of the software development process.

3.1 Overall structure of SAABNet

The variables in SAABNet can be divided into three categories:

- Architecture attributes
- Quality Criteria
- Quality Factors

This categorization was inspired by Mc Call’s quality requirement framework [12]. In this framework,

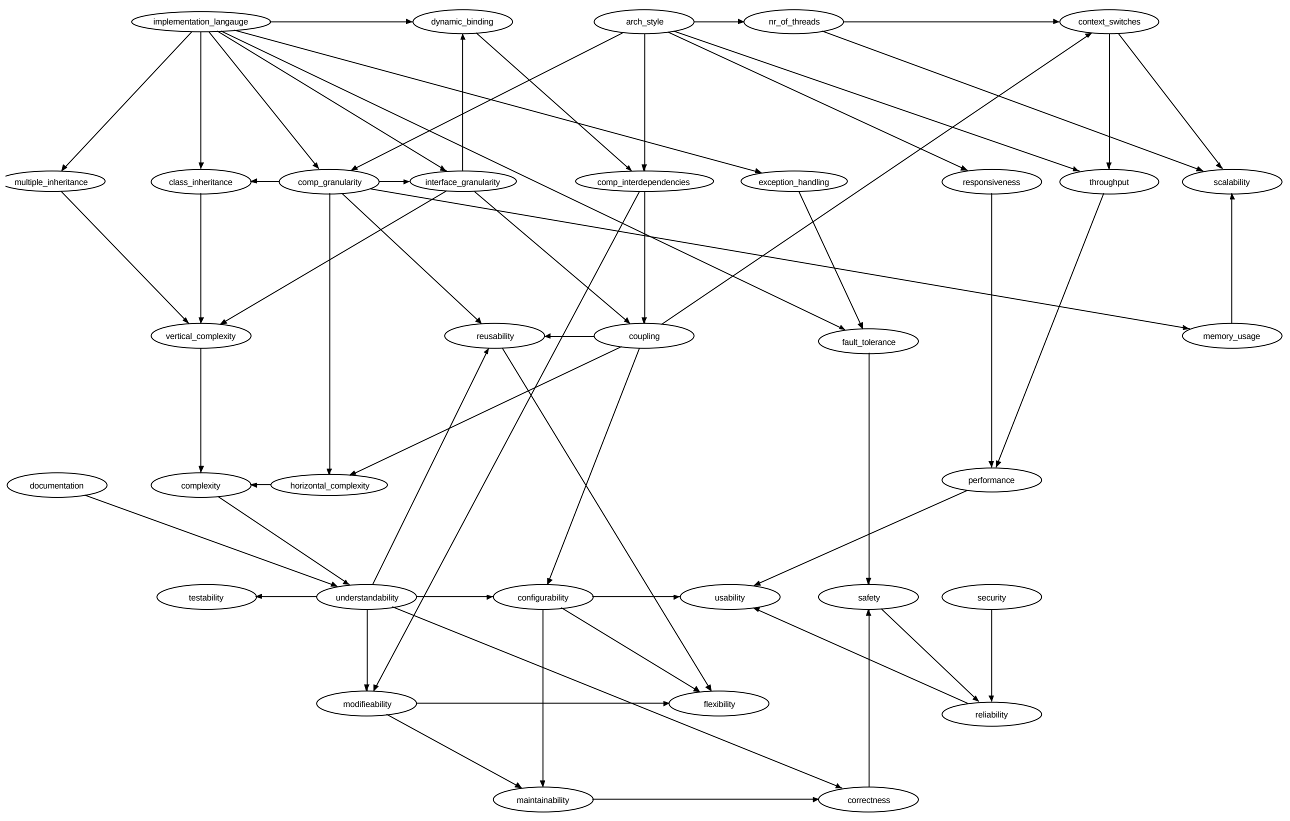

FIGURE 2. Qualitative specification of SAABNet

abstract quality factors, representing quality requirements, are decomposed in less abstract quality criteria. We have added an additional decomposition layer, called architecture attributes, that is even less abstract. Architecture attributes represent concrete, observable artifacts of an architecture.

Though this suggests a nicely layered tree for our BBN, this is not the case. SAABNet has many arrows and it is impossible to give a nicely layered view of this network. However, this layered structure does reflect the fact that variables with no outgoing arrows are more abstract than variables with no incoming arrows.

SAABNet is capable of finding non trivial argumentation (by following the arrows in the graph in the reverse direction) for the state of variables in the model. This argumentation nicely matches the argumentation that would otherwise be provided by an expert designer.

3.2 Qualitative specification

In this section we discuss the qualitative specification of SAABNet. In figure 2, a qualitative representation of our network is given (i.e. a directed acyclic graph). Though at first sight our network may seem rather complicated, it is really not that complex. While designing we carefully avoided having to many incoming arrows for each variable. In fact there are no variables with more than three incoming arrows. The reason that we did this was to keep the quantitative specification simple. The more incoming arrows, the higher the number of combinations of states of the predecessors. The cleverness of a BBN is that it organizes the variables in such a way that there are few dependencies (otherwise the number of conditional probabilities becomes exponentially large). Without a BBN, all combinations of all variable states would have to be considered (nearly impossible to do in practice because the number rises exponentially). In addition to limiting the number of incoming arrows we also limited the number of states the variables can be in. Most of the variables in our network only have two states (i.e. good and bad or high and low etc.). We may add more states later on to provide greater accuracy.

We will give a short description of the variables (in alphabetical order) in our network and indicate how they depend on other variables. We have organized them into three groups corresponding with the different types of variables we discussed in 3.1.

3.2.1 Architecture Attributes

arch_style (pipesfilters/broker/layers/blackboard).

This is the only variable with more than two states. The styles they represent correspond with architectual styles discussed in [2]. This variable has no incoming arrows so all the states have the same default probability.

class_inheritance_depth (deep/not deep). Dependencies: implementation_language, comp_granularity. This variable indicates the inheritance depth. Studies have shown that inheritance hierarchies, deeper than approximately three, are more difficult to comprehend [3]. It is important to note that this variable is not meant to be interpreted this exactly. I.e. if the inheritance hierarchy depth stays below three mostly and only incidentally is deeper than three, the architecture should have high probability for not so deep inheritance. Whether an inheritance hierarchy is deep or not is partly a design decision but it is also influenced by the programming language used. Nearly all OO languages come with some sort of framework that offers default functionality. The way this framework is structured is reflected in applications created with the language. Furthermore also the size of components is important. Large, coarse-grained components do not rely on inheritance so much so the inheritance depth is not likely to be as deep as with fine-grained components.

comp_granularity (fine-grained/coarse-grained). Dependencies: implementation_language, arch_style. This variable acts as an indicator of component size. A component, in our view, can be anything from a single class up to a large number of classes [5]. In the first case we speak of fine-grained component granularity and in the other case we speak of coarse-grained granularity. Creating coarse-grained or fine-grained components is largely a design decision. Developers have a natural tendency to concentrate functionality in groups which leads to coarse-grained components. Creating smaller components usually is the result of careful designing. This can be influenced by the implementation language (some languages provide more mechanisms to support this than others) and the architectual style which is a dominant factor in the whole architecture.

comp_interdependencies (many/few). Dependencies: arch_style, dynamic_binding. This indicates the number of dependencies between components. Dependencies can be any sort of reference to other components: it can be a hardcoded reference or it can be of a more dynamic form (for instance through an event mechanism). The type and number of dependencies are partially influenced by the architectural style. Some styles, for instance the broker style, explicitly use dynamic binding to link components together. The involved components thus have very few interdependencies. Other styles like the layers style are more performance oriented and avoid the use of (slow) dynamic binding. This leads to a more static architecture with more dependencies between the components.

context_switches (many/few). Dependencies: coupling, nr_of_threads. A context switch can occur in multi threaded systems when data currently owned by a particular thread is needed by another thread. Especially on distributed architectures this can be the cause of performance hits. But also on smaller architectures context switches can be bad for performance.

coupling (static/loose). Dependencies: interface_ granularity, comp_interdependencies. Dependencies can have several forms. This variable acts as an indicator of which type of dependency is used mostly. With static coupling, components are linked to each other with hard references while loose coupling indicates that the references are more dynamic (for instance because an event mechanism is used or because the reference to components is obtained dynamically). Interfaces are often used to add typing to loose coupled component relations while components with many interdependencies are likely to be coupled in a static way.

documentation (good/bad). This is an input variable, if no evidence is entered a probability of 0.7 is assigned to the ‘bad’ state (this reflects the situation that in most organizations documentation is not entirely adequate).

dynamic_binding (high/low). Dependencies: implementation_language, interface_granularity. Modern languages such as C++ and Java support dynamic binding (i.e. the decision what code to execute is delayed until runtime). If static binding (i.e. the relations between executable code pieces are fixed at compiletime) is used the architecture will be less flexible. The presence of interfaces and or abstract classes is usually an indicator that dynamic binding is used. Client components can than use references with the involved interface/abstract class as a type. In C++ dynamic binding has a performance impact that discourages users to apply it. Java on the other hand encourages dynamic binding by making it the default way of binding.

exception_handling (yes/no). Dependencies: implementation_language. Exception handling is a mechanism that helps to capture fault states in a program. This variable can have the states ‘yes’ (exception handling is used in the architecture) and ‘no’ (exception handling is not used). Some languages (like Java [6]) support this natively while in other languages no default mechanism is available to do exception handling. For this reason it is less likely (not impossible) that a system using such a language will have exception handling because it is more troublesome to implement.

implementation_language (java/c++). We have only included two languages (Java and C++) for this input variable in the current network. Adding more languages is not so difficult if the qualitative network is left intact. Only the quantitative part will have to be edited (all the conditional probabilities for the states in the variables on the outgoing arrows of this variable).

interface_granularity (coarse-grained/fine-grained).

Dependencies: comp_granularity, implementation_-language. In [5] we introduced a conceptual model of how to model a framework. One of the aspects of this model is to use small interfaces that implement a role as opposed to the traditional method of putting many things in a single interface. We refer to these small interfaces as fine-grained interfaces and to the larger ones as coarse-grained interfaces. This variable is an indication of whether fine-grained or coarse-grained interfaces are used in the architecture. Coarse-grained components are likely to have coarse-grained interfaces. Furthermore, some programming languages promote the use of small grained interfaces more than others.

multiple_inheritance (yes/no). Dependencies: implementation_language. Indicates whether multiple inheritance is used in an architecture. Some languages do not support multiple inheritance and even if it is supported, it is not always used. Just like a deep inheritance tree, use of multiple inheritance is bad for understandability.

nr_of_threads (high/low). Dependencies: arch_style. Indicates whether there are many threads or not. Some architectual styles make it easy to split a system into threads while with other styles this is more difficult.

3.2.2 Quality Criteria

complexity (high/low). Dependencies: vertical_complexity, horizontal_complexity. This variable indicates whether an architecture is perceived as complex. Vertical and horizontal complexity are explained in more detail below.

fault_tolerance (tolerant/intolerant). Dependencies: implementation_language, exception_handling. A fault tolerant architecture recovers nicely (in a well defined way) from run time errors. Some languages are safer than others in that they provide mechanisms to prevent certain types of fault situations. Exception handling in addition helps solve fault situations should they arrive. Of course there are other mechanisms for fault tolerance which in the current version of our network are not taken into account. If such mechanisms are used, this variable will have to be set to the appropriate value manually.

horizontal_complexity (high/low). Dependencies: coupling, comp_granularity. We split up complexity into two variables (horizontal and vertical complexity). The reason for this was that with just one complexity variable there would be many incoming arrows in the qualitative network. With horizontal complexity we mean the complexity of the relations between the components. Vertical complexity on the other hand refers to the complexity of inheritance relations (and ‘implements’ relations in Java). Horizontal complexity is usually the result of many, hardwired relations between components. This typically is the case if there are many small components that are linked together in a static way.

memory_usage (high/low). Dependencies: comp-_granularity. Indicates whether an architecture is likely to use much memory or not. Large components are usually more memory efficient than a large cluster of small components.

responsiveness (good/bad). Dependencies: arch_style. Indicates whether the application is responsive enough. This factor is influenced by the architectual style. Some styles limit the responsetime while others don’t have these limitations.

security (secure/unsecure). The system is secure if it protects itself from malicious users and prevents users from causing fault situations.

testability (good/bad). Dependencies: understandability. This variable gives an indication of whether the resulting system can easily be tested or not. At this moment in time it is only dependable of understandability but additional variables may be added later.

throughput (good/bad). Dependencies: context_switches, arch_style. Indicates whether implementations of the architecture process enough data. Context switches take time and are therefore bad for the throughput. The architectual style is another important factor since that determines how the data will flow through the system.

understandability (good/bad). Dependencies: complexity, documentation. This variable is influenced by documentation and complexity. In other words: a complex system that has poor documentation cannot be understood easily whereas a well documented system with low complexity can be understood easily.

vertical_complexity (high/low). Dependencies: multiple-_inheritance, class_inheritance_depth, interface-_granularity. Also see horizontal complexity. As opposed to horizontal complexity this variable gives an indication of the complexity of the vertical inheritance relations. Vertical complexity can be caused by deep inheritance trees but may also be the result of using multiple inheritance. Also the use of fine-grained interfaces may make the hierarchy complicated.

3.2.3 Quality Factors

configurability (good/bad). Dependencies: coupling, understandability. This indicates the ability to configure the architecture at runtime (for compile time configurability see the variable modifiability). There are two types of configurability: parameter modification and component rearranging. Both types of configuration become easier if the architecture can be easily understood. If, in addition, coupling is loose, the second type of configuration becomes more like the first type (i.e. components can be rearranged by setting certain parameters).

correctness (good/bad). Dependencies: maintainability, understandability. This variable indicates whether implementations of the architecture are likely to behave correctly. I.e. whether they will always give correct output. An architecture that is well understood and easy to fix is more likely to be correct than an architecture that isn't understood or easy to maintain.

flexibility (good/bad). Dependencies: modifiability, reusability, configurability. Flexibility is the ability to adapt to new situations. A flexible architecture can easily be tuned to new requirements and to changes in its environment. These three variables all represent a way of adapting to a new situation. The easier this is, the more flexible the architecture.

maintainability (good/bad). Dependencies: configurability, modifiability. In our network the variable maintainability depends on two other variables: modifiability and configurability. This suggests a nice definition for maintainability: the ability to change the system either by configuring it or by modifying parts of the code in order to meet new requirements. This definition can also be mapped to the decomposition of maintainability into perfective, corrective and adaptive maintenance [18]. Perfective and corrective maintenance are covered by modifiability (they both require source code changes) while adaptive maintenance is covered by configurability (no source code changes are required). Maintenance is all about performing changes in the architecture. If this is made easy (either by improving modifiability or by improving configurability), an architecture can be called maintainable.

modifiability (good/bad). Dependencies: understandability, coupling. The ability to modify an architecture. Modification can be either perfective or corrective maintenance. In order to be able to perform modifications to an architecture, the architecture has to be understood well. Loosely coupled code is easier to change because the changes have less impact on the other code.

performance (good/bad). Dependencies: throughput, responsiveness. Indicates whether the system will perform well. A well performing system has a high throughput and a good repsonsetime. Any problems in this area are reflected as bad performance in our network.

reliability (reliable/unreliable). Dependencies: safety, security. Reliability depends on security and safety in our model. So the definition of a reliable architecture we use here, is that architectures have to be both safe (functions correctly and does no harm to its environment) and secure (it can't be abused by users) in order to be reliable.

reusability (good/bad). Dependencies: understandability, comp_granularity, coupling. An architecture supports reuse if people are able to understand it, if the components sufficiently large enough to implement enough functionality but not so big they are inflexible and if the components are loosely coupled (i.e. reusing one component doesn't require the use of all other components).

safety (safe/not safe). Dependencies: fault_tolerance, correctness. Safety is all about not affecting the environment in a negative way. For software architectures this means that faults need to be taken care of in a well defined way (exception handling) and that the system gives correct output if there's is no fault situation (correctness).

scalability (good/bad). Dependencies: memory_usage, context_switches, nr_of_threads. With scalability we refer to performance scalability. I.e. the system is scalable if performance goes up if better hardware is used. While this may seem trivial, this is not true for all systems. in fact there are systems that are known to perform worse when more processors are used (the performance hit of context switching outweighs the benefit of extra processing power). To be able to use multiple processors, a system has to be multithreaded. The more threads, the easier it is to distribute them. A limiting factor is the number of context switches. If there are many contextswitches between the threads, the cost of performing them will outweigh the benefit of extra processing power. The memory consumption of the system can also be a limiting factor.

usability (good/bad). Dependencies: performance, configurability, reliability. There are a number of factors influencing the usability of an architecture: performance, configurability and reliability. These are all important factors from a user's point of view.

3.3 Quantitative Specification

Since quantitative information about the attributes we are modeling here is scarce, our main method for finding the right probabilities was mostly through trial and error. Since our assessment did not provide us with detailed information, we provided the network with estimates of the conditional probabilities. Since the goal of this network is to provide qualitative rather than quantitative information., this is not necessarily a problem.

A complete quantitative specification of our network is beyond the scope of this paper. A reason for this is that there are simply too many relations to list here. Our network contains 30+ variables that are linked together in all sorts of ways. A complete quantitative specification would have to list close to 200 probabilities. As an illustration we will show the conditional probabilities of the configurability variable in SAABNet.

understandability	*good*		*bad*	
coupling	*loose*	*static*	*loose*	*static*
good	0.9	0.2	0.7	0.1
bad	0.1	0.8	0.3	0.9

TABLE 1. Conditional probabilities configurability

Configurability depends on understandability and coupling. In table 1 the conditional probabilities for the the two states of this variable (good and bad) are listed. Since there are 2 predecessors with each two states, there are 2 times 2 makes 4 combinations of states for each state in configurability. Since we have two states that is 8 probabilities for this variable alone. Note that the sum of each column is 1.

The precision of our model is not very high. we used one decimal for the probabilities, so any output the BBN gives can't be more accurate than that. Instead of using the exact probabilities we prefer to interpret the figures as trends which can be either strong if the differences between the probabilities are high or weak if the probabilities do not differ much in value.

Possible uses of our model:

- Assistance in selecting system properties based on quality requirements. I.e. we want a flexible, highly configurable system without taking a performance penalty. Our network will suggest what states the other variables need to be in to make this likely. This does not automatically mean that other values of these variables won't give this result, it just means that it is not as likely to happen.
- Verifying of choosing certain properties for the variables indeed has the wanted effect on other variables. This is useful to provide argumentation for decisions early in the design process.
- Identifying variables that will need special attention during the development process. I.e. the BBN indicates that there will be a problem with one of the variables. Avoiding the causes for this problem may improve the situation.

Our BBN provides

- Arguments for existing ideas about the architecture
- Early warning for problem areas in the architecture
- Argumentation for taking design decisions

The BBN we discuss in this paper is a first version primarily intended to study the feasibility and usefulness of doing qualitative assessments using BBNs. If successful, SAABNet will have to go through a number of evolution cycles in which more variables are added and in which the conditional probabilities are enhanced.

4 Validation

As a proof of concept, we implemented SAABNet using Hugin [7] and used it with some cases. This tool makes it possible to draw the network and enter the conditional probabilities. It can also run in the so called compiled mode where evidence can be entered and the conditional probabilities for each variable's states are recalculated (for a complete specification of SAABNet in the form of a Hugin file, please contact the first author). The primary aim for the validation was to show that even with a simple model as SAABNet, it is possible to get useful qualitative assessments.

The evaluation method we used consists of the following steps

- Take an existing architecture
- Use assessment techniques and interviews to unveil different properties of the architecture
- Feed this information to the network and verify the findings of the network in the real architecture
- OR partially feed the information to the network and check how much of the other information is deduced by the network

For validation of the output we relied on the same sources that provided us with the input for the cases. In most cases this was system documentation.

Entered evidence		Output of the network	
documentation	bad	arch_style	layers (0.47)
class_inheritance_depth	deep	configurability	bad (0.76)
comp_granularity	coarse_grained	coupling	static (0.76)
comp_interdependencies	many	horizontal_complexity	high (0.66)
complexity	high	maintainability	bad (0.71)
context_switches	few	multiple_inheritance	yes (0.77)
implementation_language	C++	vertical_complexity	high (0.87)
interface_granularity	coarse_grained	modifiability	bad (0.90)
		reusability	bad (0.68)
		understandability	bad (1.0)

TABLE 2. Status Quo

We conducted a number of tests based on real architectures. The results of these tests are listed in table 2 till table 5. Each case is discussed in more detail in the sections after this. All tests were conducted with the same version of the network. For the output variables we picked a subset of the variables for each case. The reason for this selection is that not every variable is worth consideration in every case. Also often the network does not provide a clear preference for a variable state indicating a lack of evidence. For clarity we left these variables out. We did include variables with either confirming or conflicting values. In the discussion of the results below, conflicts with reality are discussed and clarified.

4.1 Case 1: An embedded system Architecture

For our first case we evaluated the architecture of aswedish company that specializes in producing embedded software for hardware. The devices involved, run on proprietary hardware and software. We were allowed to examine this company's internal documents for our cases.

The software, originally written in C, has been ported to C++ over the past years. Most of the architecture is implemented in C++ nowadays. The current version of the architecture has recently been evaluated in what could be interpreted as a peer review. The main goal of this evaluation was to identify weak spots in the architecture and come up with solutions for the found problems. The findings of this evaluation are very suitable to serve as a testcase for our BBN. We were allowed access to internal documents for this purpose.

4.1.1 Status Quo

The current architecture has a number of problems (which were identified in the evaluation project). In this case we test whether our network comes to the same conclusions and whether it will find additional problems.

Facts/evidence. We know several things about the architecture that can be fed to our network:

- C++ is used as an implementation language
- The documentation is incomplete and usually is not up to date
- Because of the use of frameworks, the class inheritance depth is deep.
- Components are coarse-grained
- There are many dependencies between the modules and the components
- The whole architecture is large and complicated. It consists of hundreds of modules adding up to hundreds of thousands lines of code.
- Interfaces are only present in the form of header files and abstract classes form the frameworks
- There are very few context switches (this has been a design goal to increase performance)

Based on this knowledge we can enter the evidence listed in table 2.

Output of the network. In table 2 some of the output variables are shown. The results clearly show that there is a maintainability problem. There is a dependency between configurability and maintainability and a dependency between modifiability and maintainability in figure 2. So, not surprisingly, modifiability and configurability are also bad in the results. Reusability (depends on understandability, comp_granularity and coupling) is also bad since all the predecessors in the network also score negatively. The latter, however, conflicts with the company's claims of having a high level of reuse.

In section 3.2 we described the reusability variable in an implicit way. We merely listed the prerequisites for reusability (i.e. understandability, component granularity and coupling). Clearly the architecture scores bad on these prerequisites (poor understandability, coarse-grained components and

Entered evidence		Output of the network	
arch_style	broker	configurability	good (0.52)
class_inhertance_depth	deep	maintainability	good (0.64)
comp_granularity	coarse_grained	modifiability	good (0.66)
interface_granularity	coarse_grained	reusability	bad (0.65)
context_switches	few	understandability	good (0.64)
documentation	good	coupling	loose (0.54)
implementation_language	C++	correctness	good (0.75)
		comp_interdependencies	few (0.79)

TABLE 3. Planned changes

static coupling) so the conclusion of the network can be explained. The network only considers binary component reuse. This is not how they reuses their code. Instead, when reusing code, they take existing modules, which are then tailored to the new situation. Another reason why their claim of having reuse in their organization legitimate despite the output of SAABNet is that they have a lot of expert programmers who know a great deal about the system. This makes the process of adapting code to new situations a bit easier than would normally be the case.

The network also gives the layers architectural style the highest probability (out of four different styles). This is indeed the architectual style that is used by them. As can be deduced from the many outgoing arrows of this variable in our network, this is an important variable. Choosing an architectural style influences a lot of other variables. It is therefore not surprising that it picks the right style based on the evidence we entered.

4.1.2 Planned Changes

To address the problems mentioned, the company plans to modify their architecture in a number of ways. The most important architectural change is to move from a layers based architecture to an architecture that still has a layers structure but also incorporates elements of the broker architecture. A broker architecture will make it easier to plug in components to the architecture. In addition, this will improve the runtime configurability.

Apart from architectural changes, also changes to the development process have been suggested. These changes should lead to more accurate documentation and better test procedures. Also modularization is to be actively promoted during the development process.

Facts/evidence.

- C++ is still used as a primary programming language.
- Documentation will be better than it used to be because of the process changes.
- The inheritance depth will probably not change since the frameworks will continue to be used.
- The component granularity will still be coarse-grained.
- The component interfaces will remain coarse-grained since the frameworks are not affected by the changes.
- There are still very few context switches.
- The architecture is now a broker architecture.

Output of the network. A key question is whether our network predicts the expected result of doing the suggested changes. One of the reasons the broker architecture has been suggested was that it would reduce the number of interdependencies. Our network confirms this with a high probability for few component interdependencies. However, the network does not give such a high probability for loose coupling (as could be expected from applying a broker architecture). The reason for this is that the involved components are coarse-grained. While the relations between those components are probably loose, the relations between the classes inside the components are still static.

A second reason for using the broker architecture was to increase configurability. In particular, it should be possible to link together components at runtime instead of statically linking them at compiletime. The low score for good configurability is a bit add odds with this. It is an improvement of the high probability for bad configurability in the previous case, though. The reason that it doesn't score very high yet is that the influencing factors, understandability and coupling, don't score high probabilities for good and loose. The improved documentation did of course have a positive effect on understandability but it was not enough to compensate for the probability on high complexity.

Entered evidence		Output of the network	
class_inheritance_depth	deep	complexity	low (0.62)
comp_granularity	coarse-grained	configurability	high (0.55)
comp_interdendencies	few	correctness	good (0.73)
exception_handling	yes	fault_tolerance	tolerant (0.70)
implementation_language	c++	flexibility	good (0.55)
interface_granularity	coarse-grained	maintainability	good (0.65)
memory_usage	low	modifiability	good (0.66)
multiple_inheritance	no	reliability	reliable (0.74)
		reusability	bad (0.64)
		usability	good (0.65)
		understandability	good (0.52)

TABLE 4. EPOC 32: Prediction of design goals by architectual properties

4.2 Case 2: Epoc32

Epoc32 is an operating system for PDAs (personal digital assistants) and mobile phones. It is developed by Symbian. The Epoc32 architecture is designed to make it easy for developers to create applications for these devices and too make it easy to port these applications to the different hardware platforms EPOC 32 runs on. Its framework provides GUI constructs, support for embedded objects, access to communication abilities of the devices, etc.

Since devices like PDAs and mobile phones have very strict requirements on stability and memory usage, the architecture provides mechanisms to make it easy to meet those requirements. The Epoc32 operating system was programmed in C++, which is also the main language to develop the applications in. When programming for the Epoc32 there are some restrictions on the usage of C++. Multiple inheritance, for instance, is restricted to one implementation class and possibly more than one abstract class.

To learn about the EPOC 32 architecture we examined Symbian's online documentation [17]. This documentation consisted of programming guidelines, detailed information on how C++ is used in the architecture and an overview of the important components in the system.

4.2.1 Architecture properties predict design goals

In this case we examine whether the design goals of the EPOC 32 architecture are predicted by our model given the properties we know about it. The design goals of the EPOC 32 architecture can be summarized as follows:

- It has to perform well on limited hardware
- It has to be small to be able to fit in the generally small memory of the target hardware
- It must be able to recover from errors since EPOC programs are expected to run for months or even years
- The software has to be modular so that the system can be tailored for different hardware platforms
- The software must be reliable, crashes are not acceptable

Facts/evidence. We assessed some architectual properties using the online EPOC documentation [17]. From this documentation we learned that:

- A special mechanism to allocate and deallocate objects is used
- Multiple inheritance is not allowed except for abstract classes with no implementation (the functional equivalent of the interface construct in Java).
- The depth of the inheritance tree can be quite deep. There is a convention of putting very little behavior in virtual methods, though. This causes the majority of the code to be located in the leafs of the tree.
- A special exception handling mechanism is used. C++ default exception handling mechanism uses too much memory so the EPOC 32 OS comes with its own macro based exception handling mechanism.
- Since the system has to operate in devices with limited memory capacity, the system uses very little memory. In several places memory usage was a motivation to choose an otherwise less than optimal solution (exception handling, the way DLLs are linked)
- Components are medium sized. I.e. large components are pointless because they don't provide enough modularity. Small components on the other hand have too much overhead to run efficiently. Components come in the form of one or more DLLs or an executable. The general rule is

Entered evidence		Output of the network	
configurability	good	class_inheritance_depth	not deep (0.52)
fault_tolerance	tolerant	comp_granularity	coarse-grained (0.83)
memory_usage	low	comp_interdendencies	few (0.75)
modifiability	good	exception_handling	yes (0.80)
performance	good	implementation_language	java (0.66)
reliability	reliable	interface_granularity	fine-grained (0.58)
		multiple_inheritance	no (0.77)

TABLE 5. EPOC 32: Prediction of architectual properties by design goals

that components contain a group of related classes. The presence of unrelated classes in a components may cause the component to be split into two components.

- There are few dependencies between components. In particular circular dependencies are not allowed.
- Generally components can be replaced with binary compatible replacements which indicates that the components are loosely coupled.

Output of the network.

The output of the network confirms that the right choices have been made in the design of the EPOC 32 operating system. Our network predicts that low complexity is probable, high reliability is also probable. Furthermore the system is fault tolerant (which partially explains reliability.). The system also scores well on maintainability and flexibility. A surprise is the low score on reusability. Unlike the previous case, the EPOC 32 features so called binary components. What obstructs their reuse is the fact that the components are rather large and the fact that the interfaces are also coarse-grained. Also of influence is the fifty fifty score on understandability (good understandability is essential for reuse). The latter is probably the cause of a lack of evidence, not because of an error in the network. The available evidence is insufficient to make meaningful assumptions about understandability.

4.2.2 Design goals predict architecture properties

Though its certainly interesting to see that the architectural properties predict the design goals, it is also interesting to verify whether the design goals predict the architectual properties.

Facts/evidence. In this case we entered properties that were presumably wanted features of the EPOC architecture:

- Fault tolerance and reliability are both important for EPOC since EPOC systems are expected to run for long periods of time. System crashes are not acceptable and the system is expected to recover from application errors.
- Since the system has to operate on relatively small hardware, performance and low memory usage are important
- Since the system has to run on a wide variety of hardware (varying in processor, memory size, display size), the system must be tailorable (i.e. configurability and modifiability should be easy)

Output of the network. It is unreasonable to expect our network to come up with all the properties of the EPOC 32 OS based on this input. The output however once again confirms that design choices for EPOC 32 make sense. One of the interesting things is that our network suggests a high probability on Java as a programming language. While EPOC 32 was programmed in C++, its designers tried to mimic many of Java's features (also see [17]). In particular they mimicked the way Java uses interfaces to expose API's (using abstract classes with virtual methods), they used an exception handling mechanism, they created a mechanism for allocating and deallocating memory which is safer than the regular C++ way of doing so. Considering this, it is understandable that our network picked the wrong language.

Our network also predicts coarse-grained components which is correct. In addition to that it gives a high probability for the presence of exception handling which is also correct. The net work is also correct in predicting no multiple inheritance and few component interdependencies. It is wrong, however, in predicting an low inheritance depth and predicting fine-grained interfaces. The latter two errors can easily be explained since they would only help achieving the goals that were set for the EPOC 32 architecture.

4.3 Evaluation of SAABNet

Our evaluation clearly shows that even with a simple model, as presented in this paper, it is possible to make qualitative assessments about an architecture that can rival with expert assessments. Especially in situations where little information is available (the second EPOC case for instance), useful output is given. We have been able to give a logical explanation for all incorrect output of the network. In all cases the explanation either pointed out that there was a lack of evidence or that given the input the output was generally true. The latter stresses the fact that our network gives probabilities as output. This means that sometimes reality is simply unlikely but not impossible given the limited set of facts fed to the network.

5 Related work

Important work in the field of BBNs is that of Judea Perl [16]. In this book the concept of belief networks is introduced and algorithms to perform calculations on BBNs are presented. Other important work in this area is that of Drudzel & Van der Gaag [4] where methodology for quantification of a BBN is discussed.

We were not the first to apply belief networks to software engineering. In [14] and [15], BBNs are used to assess system dependability and other quality attributes. Contrary to our work, their work focuses on dependability and safety aspects of software systems.

The qualitative network we created could be perceived as a complex quality requirement framework as the one presented by Mc Call [12]. Apart from our model being more complex, there are some structural differences with Mc Call. In our model abstract attributes like flexibility and understandability are decomposed into less abstract attributes (follow the arrows in reverse direction). Mc Call's decomposition is far more simple than ours is: it only has three layers and there are no connections within one layer. We think that his decomposition is too simplistic for our goal which is to make useful qualitative assessments about software architecture using a BBN. Mc Call's decomposition does not model independencies very well (which essential for a BBN). Many criteria like "modularity" show up in the decomposition of nearly every quality factor. In a BBN that would lead to many incoming arrows. We feel that our model may be a better decomposition because it tries to find minimal decompositions and groups simple quality criteria into more abstract ones. An example of this is our decomposition of complexity into vertical and horizontal complexity. However, continued validation is required to prove our position.

Lundberg et al. provide another decomposition of a limited number of quality attributes [9] . Like Mc Call's decomposition. Their decomposition is a hierarchical decomposition. We adopted and enhanced their decomposition of performance into throughput and responsiveness. However, we did not use their decomposition of modifiability into maintainability and configurability as we needed a more detailed decomposition. Rather we adopted Swanson's decomposition of maintenance into perfective, adaptive and corrective maintenance. We mapped the notion of perfective and corrective maintenance onto modifiability while adaptive maintenance is mapped onto configurability. A reason for this difference in decomposition is that we prefer to think of modifiability as code modifications and of configurability as run time modifications.

The SAABNet technique, we created, would fit in nicely with existing development methods such as the method presented in [1] which was developed in our research group. In this design method, an architecture is developed in cycles. After each cycle, the architecture is evaluated and weaknesses are identified. In the next cycle the weaknesses are addressed by applying transformations to the architecture. Our technique could be used to detect weak spots earlier so that they can be addressed while it is still cheap to transform the architecture.

SAABNet could also be used in spiral development methods, like ATAM (Architecture Tradeoff Analysis Method) [10], that also rely on assessments. It is however not intended to replace methods like SAAM [11] which generally require an architecture description since SAABNet does not require such a description. Rather SAABNet could be used in an earlier phase of software development.

6 Conclusion

In this paper we presented a method to automate reasoning with qualitative knowledge in the software development process. As we have pointed out, this techique should not be seen as a replacement for expert designers but rather as a assisting technique that allows a designer to work more efficiently.

Despite the small size of our belief network, we were able to get meaningful output from it in the cases we tested. In both cases, SAABNet identified both problems and positive things in the architecture. There were a few deviations from our cases though but those could be explained by either examining the network more closely or by pointing out that there was a subtle difference in perception of terminology used in SAABNet and used in the cases.

All this gives us reason to believe that a larger network based on the knowledge of experienced developers would provide us with a very powerful tool.

6.1 Future Work

The SAABNet presented in this paper represents a first version and considerable research efforts are required to mature this work. There's a lot of research being done in the field of Bayesian belief networks. Results from this research may be applicable to the domain discussed in this paper. Right now there are two developments in this field that may be applicable: object oriented BBNs and influence diagrams. OO BBNs may enhance our ability to model qualitative knowledge even better and influence diagrams may be used to support the decision process in software development. An upcoming version of Hugin [7] will have support for these enhancements. The current version has limited support for influence diagrams.

As a proof of concept, our model can be called a success, though it has to be noted that it is only a first attempt at modeling qualitative design knowledge. To make it more useful, more knowledge will have to be added to the network and the quantitative part of the network will have to be refined. Existing literature on BBNs suggest that this a very complicated task to perform.

7 References

[1] J. Bosch, P. Molin, "Software Architecture Design: Evaluation and Transformation", in Proceedings of the 1999 IEEE Coneference on Engineering of Computer Based Systems. December 1998

[2] F. Buschmann, C. Jäkel, R. Meunier, H. Rohnert, M. Stahl, "Pattern-Oriented Software Architecture - A System of Patterns", John Wiley & Sons, 1996

[3] J. Daly, A. Brooks, J. Miller, M. Roper, M. Wood, "The effect of inheritance on the maintainability of object oriented software: an empirical study", Proceedings of the international conference on software maintenance, pp. 20-29, IEEE computer Society Press, Los Alamitos, CA, USA, 1995

[4] M. J. Drudzel, L. C. van der Gaag, "Elicitation for Belief Networks: Combining Qualitative and Quantitative Information", Proceedings of the 11th Annual Conference on Uncertainty in Artificial Intelligence (UAI-95), pp. 141-148, Montreal August 1995

[5] J. van Gurp, J. Bosch, "Design, Implementation and Evolution of Object Oriented Frameworks: Concepts & Guidelines", submitted July 1999

[6] J. Gosling, B. Joy, G. Steele, "The Java Language Specification", Addison Wesley, 1996. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns - Elements of Reusable Object Oriented software", Addison-Wesley, 1995

[7] Hugin "Hugin Expert A/S - Homepage", http://www.hugin.dk

[8] Hugin, "General Information", http://www.hugin.dk/gen-inf.html

[9] L. Lundberg, J. Bosch, D. Häggander, P. O. Bengtsson, "Quality Attributes in Software Architecture Design", Accepted for the IASTED 3rd International Conference on Software Engineering and Applications, July 1999

[10] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere, "The architecture Tradeoff Analysis Method", Proceedings of ICECCS, August 1998, Monterey, CA

[11] R. Kazman, L. Bass, G. Abowd, M. Webb, "SAAM: A Method for Analyzing the Properties Software Architectures", pp. 81-90, Proceedings of ICSE 16, May 1994

[12] J. A. McCall, "Quality Factors", encyclopedia of Software Engineering, vol 2 O-Z pp. 958-969, John Wiley & Sons New York 1994

[13] Microsoft Research, "Machine Learning and Applied Statistics", http://research.microsoft.com/research/mlas/

[14] M. Neil, B. Littlewood, N. Fenton, "Applying Bayesian Belief Networks to Systems Dependability Assessment", Proceedings of Safety Critical Systems Club Symposium, Leeds, Springer-Verlag February 1996

[15] M. Neil, N. Fenton, "Predicting Software Quality using Bayesian Belief Networks", Proceedings of 21st Annual Software Engineering Workshop, 1996

[16] J. Pearl, "Probabilistic Reasoning in Intelligent Systems", Morgan Kaufmann Publishers, Inc. San Mateo 1988

[17] Symbian, "EPOC World Library", http://developer.epocworld.com/EPOClibrary/EPOClibrary.html

[18] E. B. Swanson, "The dimensions of maintenance", proceedings of the 2nd international conference on software engineering, pp. 492-497, IEEE Computer Society Press, Los Alamitos 1976