Using Evidential Reasoning to Make Qualified Predictions of Software Quality

Neil Walkinshaw
Department of Computer Science,
The University of Leicester,
UK

ABSTRACT

Software quality is commonly characterised in a top-down manner. High-level notions such as quality are decomposed into hierarchies of sub-factors, ranging from abstract notions such as maintainability and reliability to lower-level notions such as test coverage or team-size. Assessments of abstract factors are derived from relevant sources of information about their respective lower-level sub-factors, by surveying sources such as metrics data and inspection reports. This can be difficult because (1) evidence might not be available, (2) interpretations of the data with respect to certain quality factors may be subject to doubt and intuition, and (3) there is no straightforward means of blending hierarchies of heterogeneous data into a single coherent and quantitative prediction of quality. This paper shows how Evidential Reasoning (ER) - a mathematical technique for reasoning about uncertainty and evidence - can address this problem. It enables the quality assessment to proceed in a bottom-up manner, by the provision of low-level assessments that make any uncertainty explicit, and automatically propagating these up to higher-level 'belief-functions' that accurately summarise the developer's opinion and make explicit any doubt or ignorance.

Categories and Subject Descriptors

6.4 [**System Management**]: Quality Assurance

General Terms

Human Factors, Measurement

Keywords

Quality Models, Evidential Reasoning, Estimation

1. INTRODUCTION

Software quality is notoriously difficult to define and measure [1]. It is difficult to define because it is an abstract, nebulous concept that it is difficult to capture in precise terms. Furthermore, many of the underlying factors, such as code quality or usability, can be highly subjective and are largely dependent upon individual preconceptions or intuitions. Therefore, even if there is a consensus on how the various factors contribute to the final measurement, the final measurement itself can be subject to bias, and cannot be taken at face value.

Quality is commonly modelled as a decomposition into hierarchies of sub-factors – c.f. Boehm [2], Diessenboeck *et al.* [3], ISO/IEC 9126 [4] (recently revised to ISO/IEC 25010 [5]). The problem is that there is no consensus on what shape these hierarchies should take, which is indicated by the sheer number of different quality models that abound. Often, specific domains have their own, highly intricate quality models. For example the recent DO178-C standard for certifying aircraft software [6] involves 228 different quality objectives, with specific combinations that vary according to the safety-criticality of the component in question. In their study on use of software quality models in practice, Wagner *et al.* interviewed 25 practitioners [7] and recorded the use of 31 different quality models.

Even once a particular quality model has been agreed upon, there still remains the challenge of deriving a final quantitative assessment. Evidence can be qualitative and quantitative, and originate from disparate sources such as requirements reviews, test outcomes, design inspections, code metrics, user assessments, and results from verification tools. Blending such a broad spectrum of results into a single reliable conclusion about software quality relies upon individual (and invariably subjective) judgement.

These two dimensions - the multi-factorial, complex nature of software quality and the difficulty of combining and measuring these factors present a real practical barrier to assessing and communicating software quality in practice. There is little agreement on what constitutes 'software quality', there is no widely accepted means by which to aggregate and collectively interpret the various quality factors, and there is no widely accepted basis by which to allow for the subjective doubt or confidence on the part of the assessor.

Ultimately, this problem prevents the grounded, objective discussion of software quality. In an industrial context, communication about quality between developers, clients, management and other stakeholders, can be reduced to a mixture of ad-hoc metrics and intuition. Shared definitions of software quality are often implicit and poorly-defined, and subjective confidence in the various facets that lead to the assessment can be obscured. A final decision that is based on software quality in these terms is unsatisfactory because it lacks *"epistemic justification"* - there is no clear line that can be drawn from the facts and evidence to the final assessment.

The problem is well established in Software Engineering, and has been the subject of an extensive amount of work. Previous work by Fenton *et al.* and Wagner [8, 9, 7, 10] have proposed the use of Bayesian Networks to construct an explicit link between low-level quality indicators and high-level conclusions about quality. Though plausible, Bayesian Networks can be difficult to use in practice. They rely on the configuration of (potentially complex) probability tables, which rely on the availability of prior probability information. Their reliance upon prior probability information also makes it impossible to provide an meaningful assessment for a system where the assessor is ignorant of certain quality factors.

The general problem of forming high-level decisions or assessments from complex sets of factors is well established beyond the domain of software engineering. Insurance and credit companies are forced to assess an individual's life expectancy or credit-worthiness from a raft of (often dubious) metrics. In health care, the choice of a treatment programme or surgical procedure similarly comes down to a wide range of interdependent factors. The formal study of this type of problem is rooted in an area of research known as Operations Research, which dates back to the Second World War when militaries were faced with similar multi-factorial decisions with respect to troop-movement and logistics. This has given rise to several techniques, which provide a mathematical basis upon which to structure, reason about complex decision problems.

The specific problem of systematising and analysing decision problems with multiple factors or criteria is known as *Multi-Attribute Decision Analysis (MADA)* of *Multi-Criteria Decision Analysis (MCDA)* (we choose to use MADA in this paper). This paper shows how MADA techniques (specifically the Evidential Reasoning Technique [11]) can be applied to systematically assess and communicate software quality. The paper makes the following contributions:

1. It shows how the task of assessing software quality can be formalised as a MADA problem.

2. It shows how the MADA Evidential Reasoning technique can be applied, to incorporate levels of subjective doubt on the part of the assessor that may be attributed to various quality factors.

3. It walks through a small proof-of-concept, by applying the approach to the assessment of the NASA CM1 data collection and processing system for spacecraft.

4. It provides an openly-available tool that can be used with custom quality definitions.

2. BACKGROUND

The section starts with a brief overview of the relevant work on quality modelling. This is followed by a small motivating example, which will then form the basis for our case study.

2.1 Defining, Measuring, and Communicating Software Quality

'Software quality' is notoriously difficult to define [1]. Numerous quality frameworks have been proposed, which encompass factors such as 'maintainability', 'testability', 'usability' (in terms of ISO 9126), or modularity and software-system independence (in McCall's quality model). Individual quality factors can often assessed in different ways. Ultimately, software quality is multifaceted, and there is no universally agreed basis for defining or assessing it.

Given the breadth of the concept 'software quality', for our examples we focus on a slightly narrower aspect of quality assessment that is notoriously difficult to assess in its own right: the maintainability of a system. It is however important to bear in mind that the technique discussed here is not tied to this specific aspect.

The challenge of assessing maintainability has been the subject of an extensive amount of research by Diessenboeck and Wagner *et al.* [3, 10]. Maintainability is particularly difficult to assess because it can draw upon such a broad range of factors. Defining exactly what constitutes maintainability, how it can be measured, and how these measures can be combined is intrinsically difficult.

To illustrate this, we consider a relatively simple model for assessing software maintainability, based on the three dimensions that were considered by Wagner [10] in a small case-study[1]: 'Analysability', 'Quality Assurance', and 'Implementation quality'. These are all abstract concepts, so we need to break them down into more concrete notions that we are able to assess more directly. Thus 'Analysability' (referring to the ease with which the source code can be inspected) might be assessed by the size of the code-base, coupled with its complexity. 'Quality Assurance' might encompass the testability of the system and the adequacy of test sets. Adequacy is notoriously hard to assess, so we choose to break it down into code coverage and data coverage. To assess 'Implementation quality' we simply consider the quality of the code comments and identifiers.

The hypothetical hierarchy of factors discussed above is visualised in Figure 1 (ignore the bar-plots below the line – these are discussed later on). Edges are annotated with (again hypothetical) weightings for each of the different factors, showing the extent to which they contribute to parent-factors. This provides a nice illustration of one dimension of the challenge we face; how can one derive a high-level assessment of software quality from such broad range of interrelated factors of varying importance?

Finally, and perhaps most importantly, there is the issue of accounting for human doubt and ignorance. The person in charge of assessing these various factors will rarely (if ever) be 100% confident in their assessments. There may be question marks over the integrity of the various sources of evidence. They might not be sure about how to interpret a particular metric, or might not trust it enough to absolutely rely on it as a basis for assessing a given factor. Doubt and ignorance are intrinsic to any assessment of software quality. However, they tend to be ignored by conventional approaches to assessing quality, even though there

[1]We choose this model because it is (a) relatively small, forming a suitable basis for illustration, and (b) has been used as the basis for illustrating a similar quality assessment technique, with respect to the same case study that we will be using.

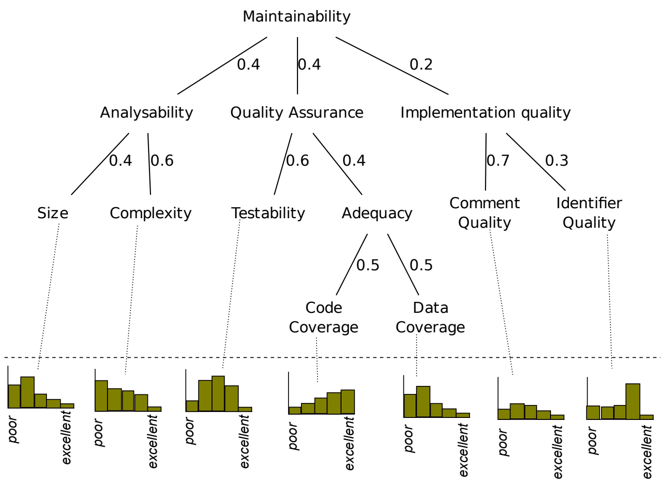

Figure 1: Hypothetical software maintainability factors (above dashed line), illustrating initial manual assessments (below the line)

is a risk that they could seriously affect (or even undermine) the overall assessment.

2.2 Motivating Example

As a motivating example, we consider the problem of assessing the quality of a software system that is intended for use on a spacecraft. Clearly, the decision of whether a piece of software is of sufficiently high quality to be deployed is particularly critical in this context. The notorious Ariane 5 explosion clearly illustrates why.

The system in question is the NASA CM1 data collection and processing system. The system has been used as a case-study for numerous software-engineering research projects, and its use here was particularly inspired by Wagner's use of the system evaluate his (similarly motivated) approach for assessing software quality [10]. Furthermore, the data has been published and is openly available, as part of the PROMISE experimental software engineering data repository [12].

Specifically, we want to assess CM1 in terms of its maintainability. The data published on PROMISE has been optimised for this, thanks to previous work by Abdelmoez [13] and Wagner [10]. The repository includes an extensive collection of code metrics for the system, along with a fully reverse-engineered UML model, along with metrics of the model.

Although the system has been extensively studied in terms of its maintainability, existing approaches have (as mentioned previously) ignored the intrinsically subjective human factor. Existing approaches tend to attempt to draw a straight line from data (i.e. code and model metrics) to conclusions about quality or maintainability. In practice however, the data tends to be subject to interpretation. Metrics only give a partial (often even a misleading) view of software quality. The relationship, for example, between cyclomatic complexity and the actual code complexity, between LOC and the size of the system, or the number of comments and code understandability, are all highly contentious. In most cases metrics can at best corroborate or inform, but it is ultimately up to the developer to interpret them.

Besides interpretation, another problem with the use of quantitative data from tools (or other developers) is the fact that the data may be unreliable. Static analysis tools can fail to parse or resolve certain relations in the code, or data that has been collected by hand might apply to a different version of the source code than the one we are assessing. This is a salient point for the CM1 system, where Shepperd *et al.* [14] have highlighted some important inconsistencies in the data-set over different studies that have utilised the data sets.

So this is our motivating scenario. How can we systematically assess the maintainability of CM1, given the knowledge available to us about its source code and model metrics? How can we assess the system in such a way that we can incorporate our doubt about the interpretation or provenance of the data? And how can we capture this information in a form that it can be readily communicated in its entirety to other interested stakeholders?

3. EVIDENTIAL REASONING

The field of MADA is very broad, and the underlying analysis problem has many variants. One popular example is the *multi-objective optimization* problem, where the challenge is to identify 'the best' solution to a problem that involves numerous factors. This is commonly solved with the help of evolutionary algorithms, and is an example of a MADA problem that has featured extensively in software testing research [15].

However, the scenario considered here is an instance of a somewhat different problem. We do not want to produce or identify a specific 'solution'. In our case all of the artefacts under analysis are already there, but we need to use them to arrive at a justifiable and transparent judgement about their quality. In our context, given the various (potentially interrelated) quality factors, which may be assessed in different quantitative or qualitative ways, we wish to arrive at a coherent decision about the overall quality of the system, which explicitly accounts for any uncertainty on our part. This is the subject of a large section of MADA research into reasoning about evidence and uncertainty. Its core theoretical framework, known as *Dempster-Schaefer Theory*, is introduced below.

3.1 Dempster-Schaefer Theory

The Evidential Reasoning technique considered in this paper (described in the next section) is based on a theoretical framework known as *Dempster-Schaefer Theory of Evidence* [16, 17, 18]. The theory is concerned with the challenge of taking subjective 'beliefs' from various sources, and to combine them in to an aggregate form, such that all of the individual sources of evidence are taken into account.

In this framework, we can take a set of factors (e.g. *system reliability*, or *value for money*). Each factor is associated with a vector of scores, often (but not necessarily) this is from 1-5 (i.e. poor to excellent) or some other suitable categorisation. This is illustrated in Table 1, which considers some possible factors that one might consider when purchasing a car. The factors of interest (value, fuel efficiency and reliability) are column headers, and the scale (poor to excellent) is used to label the rows. Each cell in the table forms a *belief statement* (e.g. "*The fuel efficiency of the car is poor*").

	Value	Fuel Efficiency	Reliability
Poor	0.0	0.1	0.1
Indifferent	0.0	0.1	0.2
Average	0.0	0.8	0.2
Good	0.6	0.0	0.1
Excellent	0.2	0.0	0.1

Table 1: Example probability distributions that attribute belief to various factors in a car purchasing decision problem.

Our subjective opinion (and doubt) about the various factors can be expressed as a 'probability distribution' over the scale for each factor. For example, as far as value is concerned, a car might be relatively cheap, but we may not be confident that it will be durable, reliable, or fuel efficient (all of which could undermine its ultimate value for money). Thus, given our uncertainty, we might be inclined to attribute it a distribution of $[0.0, 0.0, 0.0, 0.6, 0.2]$ over the categories *poor* to *excellent* - so on balance we are confident that it represents *good* value for money, but, given our ignorance, we are willing to concede that there is a small chance that it represents *excellent* value for money, pending its durability etc. Table 1 shows an example of how what some probabilities might look like for the other factors.

The term 'probability distribution' is used in quotes because it is not a true probability distribution in the Bayesian sense. In Dempster-Schaefer theory, this distribution is actually referred to as a 'Belief Function' – a term that will be also be adopted throughout the rest of this paper. The crucial difference is that, unlike Bayesian probability distributions, the sum of probabilities does *not* need to add up to 1. This is to allow for a more genuine weighting for each category in the scale. For example, for the *value* factor in Table 1, we may be convinced of our scores of 0.6, and 0.2. What about the remaining 0.2? In Bayesian probability, under the Principle of Indifference this would be evenly divided across Poor, Indifferent, and Average, giving each a weighting of 0.067. However, in Dempster-Schaefer theory, this weight is simply left as "unknown". The benefit (especially with respect to our domain of reasoning about software quality) is that this draws a firm line between information that is confidently 'known', and facts about which we are ignorant or dubious.

Ultimately, given a set of beliefs (which can be structured in this tabular form), Dempster-Schaefer theory is concerned with combining the different beliefs across the different factors. This can be used to derive high-level beliefs about the system that take the various factors into account. With respect to Table 1, given the belief-functions over the various factors, we can combine them to produce a high-level combined belief function that conveys to us the quality of the system as a whole. Over the past 40 years numerous different belief-combination approaches have been developed to do so. One such approach, Evidential Reasoning, extends the Dempster-Schaefer theory and is presented in more detail below.

3.2 Evidential Reasoning

Most realistic decision processes are too complex to neatly fit into the tabular format presented above. As discussed previously, concepts such as software quality can be hierarchical in nature. It is also unrealistic that each factor is given an equal importance. For example, a car buyer might place an emphasis on reliability over value. This adds yet another dimension to our assessment - we need to incorporate not only the different degrees of belief for various factors, but also their different respective weightings.

This delineates the problem considered by Evidential Reasoning, and reflects the illustration in Figure 1. Let us suppose that we have a multi-levelled decision problem, where different factors are combined according to weights. Let us suppose that each of the lowest-level factors is associated with its own Belief Function (shown below the dashed line) that denote their individual assessments. How can we aggregate these different beliefs to draw a meaningful conclusion about the system as a whole?

Evidential Reasoning (ER) was developed by Yang *et al.* [11]. It is an algorithmic approach to evidence-based reasoning that is founded upon Dempster-Schaefer Theory, which can process complex hierarchical decision problems of the type shown in Figure 1. The following notation and definitions are closely based on those provided by Yang *et al.* [11].

To introduce the basic notation, we start by considering a simple 2-level hierarchy, with the general attribute at the top (e.g. Maintainability), which is composed of a set E of l lower-level attributes, so $E = \{e_1, \ldots, e_l\}$. The weights of the attributes are denoted $w = \{w_0, \ldots, w_l\}$, where w_n is the relative weight of e_n and $0 \leq e_n \leq 1$. Each attribute is assessed according to a set of g grades $H = \{H_1, \ldots, H_g\}$ (e.g. the scale from 1 to 5). Thus, the user's assessment for a basic attribute e_i (denoted $S(e_i)$) can be represented as the following distribution:

$$S(e_i) = \{(H_n, \beta_{n,i}), n = 1, \ldots, g\} \quad i = 1, \ldots, l \qquad (1)$$

Here, $\beta_{n,i}$ denotes the degree of belief that attribute e_i is assessed to the grade H_n. It is always the case that $\beta_{n,i} \geq 0$ and $\Sigma_{n=1}^{N} \beta_{n,i} \leq 1$. In other words, the sum of the belief functions for an attribute can be less than 1; a degree of ignorance or even total ignorance is permitted.

The Evidential Reasoning problem is to, for some hierarchy of factors, take a set of belief functions corresponding to the leaf-nodes, and to propagate and combine those beliefs (obeying the weights w) to produce a single high-level belief function $\beta_n (n = 1, \ldots, g)$ at the root of the hierarchy, which fairly aggregates the assessments for all of the attributes e_i where $i = 1, \ldots, l$. With respect to Figure 1, the challenge is to produce a probability distribution for the general attribute 'Quality' that aggregates all of the evidence from the lower-level attributes.

3.2.1 *The Evidential Reasoning Algorithm*

The ER algorithm [11] provides a means by which to combine the belief functions for attributes. Given a hierarchy of attributes, it operates from the bottom-up, combining lower-level attributes, and propagating the belief functions up to the top, to yield a general belief-function. Most of the following description shows how a set of lower-level attribute assessments can be combined. This will be followed by a brief description of how this can be extended to a hierarchy.

The high-level process of processing a hierarchy is shown in Algorithm 1. The entry-point is the `beliefFunction` function. It is given as input the root-node of our tree n, a

```
   input: $node, InitBeliefs, w, l$
   /* $node$ is the current node in the belief-tree (initially
      this is the root-node)                                   */
   /* $w$ is a weighting for each node (in terms of its
      contribution to its ancestor)                            */
   /* $l$ is the number of levels that form a belief-distribution
      (usually 5)                                              */
1  beliefFunction($n, InitBeliefs, w, l$) begin
2      if $isLeaf(n)$ then
3          return $InitBeliefs(n)$;
4      else
5          return computeBeliefs(descendants($n$),
           $InitBeliefs, w, l$));
6      end
7  end
   input: $Nodes, InitBeliefs, w, l$
   /* $Nodes$ is a list of (sibling) nodes, where the beliefs
      are to be aggregated.                                    */
   /* $w$ and $l$ are as defined for beliefFunction            */
8  computeBeliefs($Nodes, InitBeliefs, w, l$) begin
9      for $i = 1 \to |Nodes|$ do
10         $\beta_i \leftarrow$ beliefFunction($n, InitBeliefs, w, l$);
11     end
12     return aggregateBeliefs($\beta, w, l$);
13 end
```

Algorithm 1: Functions `beliefFunction` and `computeBeliefs`, which together recursively carry out a depth-first traversal of the tree of factors, propagating belief functions from the bottom-up.

mapping from the leaf nodes of the tree to their respective belief functions $InitBeliefs$, a mapping from all nodes to their respective weightings w (where the weights of a set of siblings sum to 1), and the scale used to assess nodes l (usually 1-5). Here, $InitBeliefs$ corresponds to the definition given in Equation (1).

The process essentially consists of a depth-first traversal of the tree. The base-case is that the node is a leaf-node (line 2), which means that the user-supplied belief function can simply be returned from $initBeliefs$. Otherwise the `computeBeliefs` function is called (line 5), which is responsible for computing a belief function from the belief functions of its child-nodes (passed as a list of descendants, identified by the `descendants` function). The `computeBeliefs` function starts by iterating through the given node list $Nodes$, obtaining their respective belief functions (this is the point of recursion, with call to `beliefFunction`). Once the set of belief functions have been collected for all sibling nodes (possibly by recursion), the functions are combined by passing them to the `aggregateBeliefs` function.

The `aggregateBeliefs` function represents the core of the ER approach. This is where the belief functions of multiple sibling nodes are combined into one, overall belief function. The generated belief function is slightly different in nature to the belief functions supplied in $initBeliefs$ and described in Equation (1). It not only contains a value for each level 1 to l. It also contains an extra bar, which aggregates the doubt or ignorance - where belief functions provided in $initBeliefs$ did not sum up to 1, which implies an implicit element of doubt on the part of the user.

Given the intricacy of the process, there is not enough space here to describe it in detail. A detailed description is available in the paper by Yang *et al.* [11]. To be self-contained, the algorithm is presented in Appendix A (in a procedural form, which is intended to make it easier to read in a summary form than the declarative form used by Yang). Where possible, in-line comments have been used to at least provide an intuition of its workings.

At a high-level, this depth-first process of identifying belief-functions and combining them propagates the values that are assigned to the leaf-nodes up the tree, combining them in the process. In the process, any levels of doubt or ignorance are made explicit in the additional bar computed by `aggregateBeliefs`. This culminates in an aggregate belief function at the root of the tree, which combines all of the belief functions from the leaf-nodes, incorporates the weightings for individual nodes at each level, and makes the level of doubt explicit.

3.2.2 Illustration

The tree processing procedure can be illustrated with respect to the small example in Figure 1. We start by passing the root node ('Maintainability') to `beliefFunction`. Since this is not a leaf node, it calls computeBeliefs to combine the belief-distributions of its children. In processing the child-nodes (lines 9-10) the belief-function for each node is retrieved with a recursive call to `beliefFunction`. Starting with the left-most node ('Analysability'), this is a non-leaf node, which prompts another call to `computeBeliefs`, passing its child-nodes (this illustrates the depth-first processing of the tree).

Since all of these are leaf-nodes, their belief-functions are combined by the `aggregateBeliefs` function to produce a new, aggregated belief-function for 'Analysability'. Having retrieved this belief function, the `computeBeliefs` function moves on to retrieve the belief function for its next sibling node 'Quality Assurance'. This, being a non-leaf node, prompts a further depth-first search-aggregation process until the lower level belief functions have been combined to produce its belief-function. This process continues until all three nodes have their (aggregated) belief functions. Finally these are combined to produce the final overall belief function for 'Maintainability'.

4. APPLYING EVIDENTIAL REASONING TO SOFTWARE QUALITY

At the core of software quality lies the challenge of assimilating a wide range of sources of evidence about disparate factors into a single coherent assessment. Judgements about individual factors are often subjective, and can incorporate a certain degree of doubt or ignorance. As such, it would seem to be an ideal fit for Evidential Reasoning.

ER enables the abstract notion of quality to be broken down into its constituent parts and to quantify the relative weights that each part plays in the overall conclusion. It enables the precise specification of the extent to which there is any doubt or ignorance associated with assessments of individual factors, and to factor these in to the final high-level conclusion. Consequently, high level assessments of software quality can be formalised, weighted, and appropriately modulated according to the subjective certitude or doubt on the part of the assessor.

There are three high-level steps that are required for the application of ER to assess software quality. First of all, it is necessary to encode our interpretation of 'software quality' as a weighted hierarchy (as illustrated above the dashed line in Figure 1). Secondly, we have do decide upon a rating scale against which we wish to assess all of the factors. Finally, we have to provide our low-level belief-functions that indicate our subjective (and potentially dubious) assessments of the lowest-level elements in our quality hierarchy. These steps are discussed in their respective subsections below.

4.1 Developing or Selecting a Hierarchy of Quality Factors

As discussed previously, software quality is intrinsically hierarchical [2, 19, 3, 10]. Existing quality standards and other literature in the area tend to adopt the approach of starting from the high-level notion of quality and gradually decomposing it into more concrete notions that are individually more straightforward to understand and assess. When it comes to deciding what the specific factors are, and how they are related to each other, there is little agreement [3, 7]. Accordingly, this question is left open in this paper. The approach presented is not tied to any specific hierarchy.

If there is no existing quality definition in place, then the developer can develop their own. We have illustrated this by developing our own illustrative hierarchy in Figure 1. Of course, this hierarchy is small, focussing solely on maintainability, and is merely for the sake of illustration. In practice, a model could be more expansive, with more levels. An alternative situation is that the host organisation has adopted an existing definition of software quality (e.g. the ISO/IEC 9126 standard). If this is the case, then the factors have already been provided for us. All that remains is to attribute to them their respective weights (i.e. the relative extent to which sibling factors contribute to the assessment of their parent). In the absence of any opinions about relative weightings, they are evenly weighted by default.

It is important to emphasise that the the definition of software quality is not affected by the availability (or lack of) evidence. If, for example, a quality hierarchy definition includes test coverage, this forms a part of the calculation of software quality, *even if there is no test coverage data available.* This is an important distinction from typical approaches to assessing software quality, which are forced to tailor their definitions of software quality to the available, measurable data. With the ER approach, if we are ignorant about a particular factor, it can still be included in the form of belief functions that are all zero (signifying total ignorance). This means that the same definitions of quality can be used across a range of systems; absence of evidence is simply reflected in an increased “ignorance score” for the final result.

4.2 Selecting a rating scale

The ER approach relies on a scale that can be uniformly applied to all factors. The scale should broadly range from a low value, indicating a negative assessment, to a high value indicating a positive one. The scale adopted in this paper is 1-5, where 1 = ‘poor’ and 5=‘excellent’. However, one might choose to make the scale more or less granular, depending on the level of detail at which the quality of the system is to be assessed.

4.3 Deriving Initial Belief Functions

Of course, what constitutes ‘poor’ and ‘excellent’ is, depending on the factor in question, intrinsically subjective. The final, and most complicated step in the process is to elicit this opinion from the assessor for each of the factors, in the form of a ‘belief function’.

A belief function is produced every factor at the lowest level of the hierarchy. For each of these, the assessor has to combine: (1) Their (potentially ad-hoc) analysis of the relevant artefacts. (2) Quantitative measurements in the form of code or model metrics. (3) Their understanding of how these various sources of evidence relates to the factor in question.

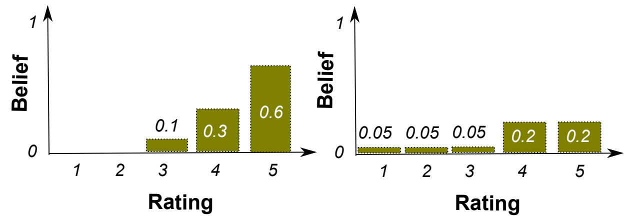

Figure 2: Examples of two belief functions. The values on the x-axis range from 1 (poor) to 5 (excellent).

Two examples of possible belief functions are shown in Figure 2. These have two dimensions: the rating scale and the ‘belief’ scale. The rating scale is as discussed above, and the levels are arranged on the x-axis. The attribution of belief values to different ratings represents the subjective balance of opinion of the assessor. The sum of the belief values cannot exceed 1 (but does not have to sum up to 1 either). If the belief values sum up to 1, this indicates that the developer is absolutely certain about the balance of their opinion. If not, it indicates that there is a degree of doubt [11].

The example on the left in Figure 2 represents an instance where the assessor is reasonably certain that the factor in question should be rated ‘excellent’ or ‘good’ (with a bias towards ‘excellent’). The assessor may have an in-depth knowledge about the system, the various sources of evidence, and their relation to the factor, and be reasonably confident about the assessment. They do however concede that there is the chance that they have misinterpreted the evidence. They might be aware that the metrics, though positive, can still be misleading, so they must accept that there remains a very small possibility that the factor could be merely be ‘adequate’, despite the overwhelming evidence to the contrary.

The example on the right illustrates a different possible scenario. The assessor is highly doubtful about how to assess this factor. There might not be any metrics available, and they might be forced to resort to their intuition, or background knowledge. From this, they have an inkling that it may be good or excellent, but admit that there is a chance that the factor could also be assessed as very poor. Ultimately, they are simply unsure. This is reflected by the fact that all of the belief values are very low, and fail to add up to 1.

5. ILLUSTRATIVE CASE STUDY

This section provides a proof-of-concept illustration of how ER can be applied to assess software quality. It is to be emphasised that this section does not seek to provide an empirical evaluation. This would necessitate a large number of software systems and human assessors, and is part of our ongoing work. Instead, this section acts as a proof-of-concept.

The section will illustrate how, in spite of varying degrees of uncertainty and ignorance, evidence and opinion about quality can be processed in a systematic way to produce a comprehensive assessment. The result is not just a single number or decision, but a tree. This illustrates the value of being able to incorporate ignorance and uncertainty, alongside clear links from evidence to high-level judgements about software quality.

5.1 Case Study and Implementation

For our case study, we assess the NASA CM1 system described in Section 2.2 in terms of its maintainability. As sources of 'evidence', we rely entirely on the data about the CM1 system that are available on the PROMISE repository [12]. This includes a table[2] that evaluates each of the 327 source code modules according to 37 source code metrics.

To begin with, we must identify a suitable hierarchy of factors that are deemed to constitute 'maintainability'. For this we adopt our example hierarchy, shown in Figure 1, and also adopt the weightings attributed to the various factors. We also choose the rating 1-5, as this is commonly used and intuitive (c.f. Likert scales).

The next step is to produce the 'belief functions' for the leaf-nodes (i.e. to replace the illustrative plots below the dashed line in Figure 1 with actual data). The belief-functions are listed below, along with some of the underlying thought processes that were used to generate them. Space restrictions prevent a detailed treatment of each factor, but this should still suffice to provide an intuition of the general process.

- **Size:** For this we consider the non-blank non-comment LOC, and Halstead Length metrics. Most of the modules have reasonably small numbers (i.e. < 10 LOC and < 500 Halstead), but there are about 20 modules where both LOC and Halstead metrics spike up. Together these give us a reasonably good picture of size, so we propose the following belief function: $[0.0, 0.0, 0.35, 0.5, 0.1]$.
- **Complexity:** The type of complexity considered here is not specified in the quality model, and is therefor somewhat ambiguous. We opt to focus on the code complexity, for which there are several metrics at our disposal, including Cyclomatic Complexity and Halstead Difficulty. When viewed on a scatter-plot these are all roughly correlated with each other. Looking more closely at Cyclomatic Complexity (which is somewhat simpler), there seems to be a significant degree of complexity. 74 of the modules have a Cyclomatic Complexity > 10 (three > 70). It is also acknowledged that complexity is notoriously difficult to assess with metrics (e.g. these metrics focus almost exclusively on logical complexity, as opposed to data complexity). As a result we propose the following belief function: $[0.2, 0.4, 0.3, 0.0, 0.0]$.
- **Testability:** To gauge testability, we look at the branching structure, by looking at the number of decision points with multiple conditions, and the Parameter Count. From these the testability seems to be quite poor. Although the average parameter count tends to be quite low (around 2), there are 30 instances with 4 or more parameters. The multiple-condition decision count is quite high (average 10.4), with several spikes (in one case up to 124). This leads to the following function: $[0.25, 0.4, 0.25, 0.0, 0.0]$.
- **Code coverage and data coverage:** We do not have any access to the test coverage information for this system. Thus, we can say nothing about them, which leads to the following belief function for both: $[0.0, 0.0, 0.0, 0.0, 0.0]$.
- **Comment Quality:** The only useful metric available is called "Percentage Comments", indicating the proportion of the module that is dedicated to comments. This is generally quite high (an average of 30%), so we can be inclined to infer that a large proportion of comments indicates that the quality is relatively high. Of course, we cannot know this. We hedge our bets by tilting our confidence towards good / excellent, but we leave 0.2 of our 'belief mass' unassigned, indicating our doubt $[0.05, 0.05, 0.1, 0.3, 0.3]$.

[2]Specifically, we use the metrics file that has been preprocessed by Shepperd *et al.* [14], who have attempted to remove implausible or erroneous data.

```
((comment_quality:1)implementation_quality:0.2,
  (testability:0.6,(code_coverage:0.5,
  data_coverage:0.5)adequacy:0.4)
  quality_assurance:0.4,(complexity:0.6,size:0.4)
  analysability:0.4)maintainability;
5
comment_quality 0.05 0.05 0.1 0.3 0.3
code_coverage 0 0 0 0 0
data_coverage 0 0 0 0 0
testability 0.25 0.4 0.25 0 0
complexity 0.2 0.4 0.3 0 0
size 0 0 0.35 0.5 0.1
```

Figure 3: ERTool input for CM1 example.

To compute the belief functions, we have developed an openly-available proof-of-concept tool (*ERTool*)[3]). This takes as input a three-part text file. The first part contains a description of the tree and the weights of the branches, using the Newick format[4]. This is followed by a number, denoting the number of elements in the rating scale. Finally, this is followed by a list of belief functions for the leaf-nodes of the tree. As output it produces a hierarchy of bar plots, an example of which is shown later. For the case study, the input is shown in Figure 3.

This illustrates the simplicity of the approach. The full quality model and all of the necessary user input can be captured in 12 lines of text.

5.2 Result and Discussion

The ER algorithm produces the final hierarchical assessment, shown in Figure 4 - a tree of 'belief functions'. Each belief function describes precisely our assessment of the various factors. The leaf-nodes simply reflect our initial scores discussed above. However, things become more interesting as different assessments are folded together. The ER algorithm combines assessments from the bottom-up, obeying

[3]https://bitbucket.org/nwalkinshaw/ertool/overview
[4]http://en.wikipedia.org/wiki/Newick_format

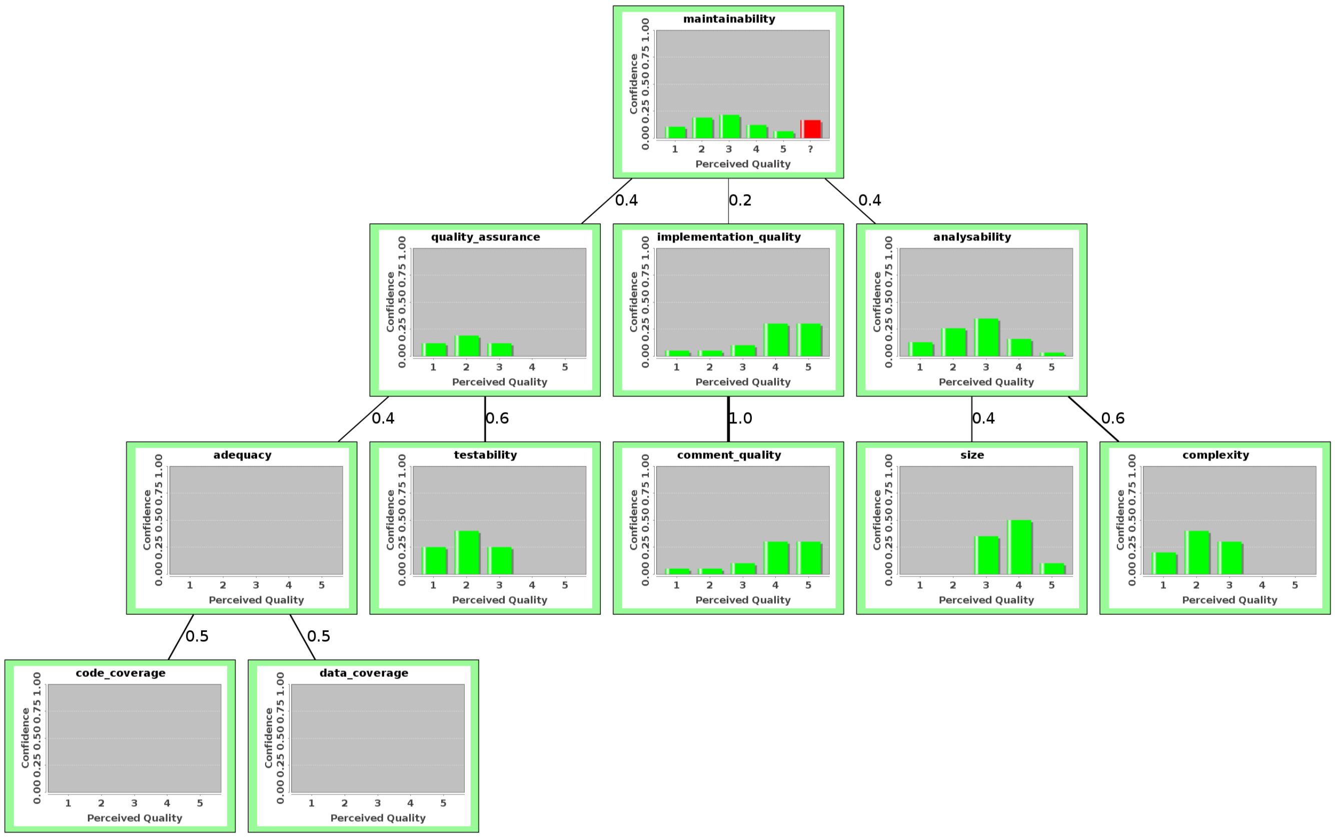

Figure 4: ER-assisted Assessment of Software Maintainability for CM1

the different weightings that are attributed to different factors in the process. This results in a full picture of the assessor's opinion about all of the factors that (they consider) to contribute to software maintenance. Although the behaviour of the ER algorithm can be difficult to discern from an algorithmic perspective, several of its behavioural properties are nicely illustrated here.

Looking at the combination of Size (skewed towards the positive end of the ratings), and Complexity (skewed towards the negative end), the result (Analysability) roughly centres on the median point, tapering off towards both extremes.

The combination of Adequacy and Testability is interesting, because Adequacy offers no new information. When combined with Testability, the result retains the same characteristics of the Testability function, but the magnitudes of the bars are substantially attenuated.

Finally, the overall assessment at the top of the hierarchy is accompanied by a special bar, denoting the doubt inherent in the overall assessment. This is computed from those belief functions where the total sum of beliefs did not add up to 1 (implying a degree of uncertainty or ignorance).

There remains the big question of whether the final result in Figure 4 is sensible and valid. From the context adopted in this paper, given the lack of prior knowledge about the system and restriction to the selected metrics, it makes sense. The metrics (especially for complexity and testability) are not favourable, which leads to a relatively low quality score around the 2-3 mark. However, at the same time there is a lot of doubt. The total ignorance about test outcomes and relatively poor alignment between the available metrics and other quality factors contribute to an overall 'doubt score' that suggests that this assessment is highly dubious.

This explicit characterisation of doubt is perhaps the most useful characteristic of Evidential Reasoning. Alternative approaches that combine metrics would provide a single, unqualified figure, which would be wide open to (mis-)interpretation. However, in this case, the assessment provides a clear and complete picture of the author's subjective assessment of CM1, replete with doubt and ignorance.

It is important to re-iterate that this section does not seek to empirically validate the use of ER in any way. One can safely assume that, with more contextual information about the developers and development processes involved etc., a completely different picture of software quality might emerge. However, the important point is that this assessment accurately captures the author's subjective assessment of the system, when limited to reasoning about a relatively restricted set of metrics of potentially questionable provenance [14]. As will be discussed later, the construction of a more systematic validation study is a part of our ongoing and future work.

6. RELATED WORK

The problem of incorporating uncertainty and subjectivity to the assessment of software quality is well-established. So far, the main proposed approach to addressing this has been to recode quality models as Bayesian networks, and to use their capacity to represent probability and uncertainty. This approach has been espoused by the work of Fenton *et al.* [9, 8], and has recently been extended by Wagner [10].

The work presented in this paper is especially comparable to the Bayesian network approach presented by Wagner. Indeed, the use of the CM1 data to validate our approach was partly inspired by his evaluation on the same data [10] (albeit with respect to slightly different quality models), to provide insights into the relationships between the two approaches.

On the surface, the two approaches are highly similar in nature. Both deal with a tree-structured quality model (though this need not necessarily be the case for Bayesian models). Both feed evidence into the bottom layers, in the form of distributions that approximate the confidence of the assessor. In both cases the data is propagated up to form an overall assessment. The interpretation of the metrics data that is used to formulate the low-level distributions is very similar.

The key difference between the approach presented here and Bayesian network approaches is, ultimately, that this approach is not based upon Bayesian statistics [18]. It will still produce a valid assessment, even if there are *no prior distributions*, or there is no evidence for particular factors. This allows the approach presented here to distinguish between *uncertainty* and total *ignorance*. If we do not know about a factor (such as the test-coverage for CM1), but it is still a part of our quality model, this ignorance is made explicit in the final assessment.

There does not seem to be an obvious way in which such ignorance could be factored in to a Bayesian model. Bayesian models rely on the (often unrealistic) premise that every eventuality can be attributed with a probability. When no prior information exists, the only option is to resort to the Principle of Indifference, by providing a 'flat' probability distribution in which all eventualities are equally possible. This is however artificial and can lead to misleading models; the coercion of probabilities into a distribution despite the the absence of any prior knowledge or evidence can end up obscuring ignorance. ER models on the other hand make any ignorance explicit. If a model provides a positive quality assessment, but there is an absence of evidence or prior distributions for many of the factors, this knowledge is made explicit in the final outcome.

7. CONCLUSIONS AND FUTURE WORK

This paper has shown how, in principle, the Evidential Reasoning approach can be applied to reason about software quality. It enables developers to define their own quality models (or to use existing ones), and allows for uncertainty when these factors are assessed. Moreover, in contrast to existing Bayesian approaches, it also distinguishes between mere uncertainty and complete ignorance. Regardless of the quality model and the available knowledge or evidence, it is always capable of producing an assessment that makes any areas of doubt and ignorance explicit, thus maintaining the integrity the final model.

As a proof-of-concept, the approach has been developed into a small, openly available tool. This was used to provide a high-level assessment of the NASA CM1 module, albeit one that was restricted to the data available from the PROMISE repository, and with respect to a small quality model. The simplicity of the approach is indicated by the amount of input required for the tool; the full model and all of the probability distributions require only 12 lines of text in this paper.

This paper does not provide a full validation of the approach. This would require feedback from developers, covering a sufficiently large number of systems. Producing this more comprehensive evaluation forms the core part of our ongoing and future work.

Acknowledgement

The author thanks Marc Roper at the University of Strathclyde and the anonymous reviewers who provided helpful feedback on earlier drafts of this paper.

8. REFERENCES

[1] B. Kitchenham and S. Pfleeger, "Software quality: the elusive target [special issues section]," *Software, IEEE*, vol. 13, no. 1, pp. 12–21, 1996.

[2] B. Boehm, J. Brown, H. Kaspar, M. Lipow, G. MacLeod, and M. Merrit, *Characteristics of software quality.* North-Holland Publishing Company, 1978, vol. 1.

[3] F. Deissenboeck, S. Wagner, M. Pizka, S. Teuchert, and J. Girard, "An activity-based quality model for maintainability," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on.* IEEE, 2007, pp. 184–193.

[4] "Iso/iec 9126-1, software engineering – product quality – part 1: Quality model," 2001.

[5] "Iso/iec 25010, systems and software engineering – systems and software quality requirements and evaluation (square) – system and software quality models," 2011.

[6] "Rtca/do-178c "software considerations in airborne systems and equipment certification"," 2010.

[7] S. Wagner, K. Lochmann, S. Winter, A. Goeb, and M. Klaes, "Quality models in practice: A preliminary analysis," in *ESEM*, 2009, pp. 464–467. [Online]. Available: http://doi.acm.org/10.1145/1671248.1671308

[8] N. E. Fenton, M. Neil, and J. G. Caballero, "Using ranked nodes to model qualitative judgments in bayesian networks," *IEEE Trans. Knowl. Data Eng*, vol. 19, no. 10, pp. 1420–1432, 2007. [Online]. Available: http://dx.doi.org/10.1109/TKDE.2007.1073

[9] M. Neil, B. Littlewood, and N. Fenton, "Applying bayesian belief networks to system dependability assessment," in *Safety-Critical Systems: The Convergence of High Tech and Human Factors: Proceedings of the 4th Safety-critical Systems Symposium Leeds, UK 6-8 February 1996*, F. Redmill and T. Anderson, Eds. Leeds, UK: Springer, 1996, pp. 71–94.

[10] S. Wagner, "A bayesian network approach to assess and predict software quality using activity-based quality models," *Information & Software Technology*, vol. 52, no. 11, pp. 1230–1241, 2010. [Online].

Available:
http://dx.doi.org/10.1016/j.infsof.2010.03.016

[11] J.-B. Yang and D.-L. Xu, “On the evidential reasoning algorithm for multiple attribute decision analysis under uncertainty,” *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, vol. 32, no. 3, 2002.

[12] T. Menzies, B. Caglayan, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan. (2012, June) The promise repository of empirical software engineering data. [Online]. Available: http://promisedata.googlecode.com

[13] W. Abdelmoez, K. Goseva-Popstojanova, and H. Ammar, “Maintainability based risk assessment in adaptive maintenance context,” in *2nd International Predictor Models in Software Engineering Workshop (PROMISE 2006), Philadelphia, PA*, 2006.

[14] M. Shepperd, Q. Song, Z. Sun, and C. Mair, “Data quality: Some comments on the nasa software defect data sets,” submitted.

[15] K. Lakhotia, M. Harman, and P. McMinn, “A multi-objective approach to search-based test data generation,” in *Proceedings of the 9th annual conference on Genetic and evolutionary computation.* ACM, 2007, pp. 1098–1105.

[16] A. P. Dempster, “A generalization of Bayesian inference,” *Journal of the Royal Statistical Society, Series B*, vol. 30, pp. 205–247, 1968.

[17] G. Shafer, *A Mathematical Theory of Evidence.* Princeton University Press, 1976.

[18] J. Y. Halpern, *Reasoning about Uncertainty.* Cambridge, Massachusetts: The MIT Press, 2003.

[19] J. McCall, P. Richards, and G. Walters, *Factors in Software Quality.* NTIS Springfield, 1976.

Appendix A: Aggregating Belief Functions

```
input: $\beta, w, l$
/* $\beta$ is the list of belief functions. $\beta_i$ refers to the
   belief function for node $i$. $\beta_{n,i}$ refers to the specific
   ‘belief-value’ for node $i$ at level $n$.                          */
/* The other parameters are described in Algorithm 1                 */
aggregateBeliefs($w, l, \beta$) begin
    // The following loop aggregates the belief masses.
    for $i = 1 \rightarrow (|\beta| - 1)$ do
        for $n = 1 \rightarrow l$ do
            $m_{n,i} \leftarrow w_i \beta_{n,i}$;
            // Calculate basic belief-masses
            if $i = 1$ then
                $c_{n,i} \leftarrow m_{n,i}$;
                // For initial computation of $k$
            end
        end
        $\bar{m}_{H,i} \leftarrow 1 - w_i$;
        // Calculate the remaining weight (not attributed to
           $\beta_i$)
        $\tilde{m}_{H,i} \leftarrow w_i \left(1 - \sum_{n=1}^{l} \beta_{n,i}\right)$;
        // Calculate the level of ignorance (unassigned
           Belief-mass) for node $i$
        $m_{H,i} \leftarrow \bar{m}_{H,i} + \tilde{m}_{H,i}$;
        $k = \left[1 - \sum_{t=1}^{l} \sum_{\substack{j=1 \\ j \neq t}}^{l} c_{t,i} m_{j,i+1}\right]^{-1}$;
        // Compute normalizing factor
        for $n = 1 \rightarrow l$ do
            $c_{n,i+1} \leftarrow$
            $k(m_{n,i} m_{n,i+1} + m_{n,i} m_{H,i+1} + m_{H,i} m_{n,i+1})$;
            // Compute combined belief mass for level $n$,
               adding node $i + 1$ to aggregated beliefs
        end
        $\bar{c}_{H,i+1} \leftarrow k(\bar{m}_{H,i} \bar{m}_{H,i+1})$;
        // Compute remaining weight after adding node $i + 1$
           to aggregated beliefs
        $\tilde{c}_{H,i+1} \leftarrow k(\tilde{m}_{H,i} \tilde{m}_{H,i+1} + \bar{m}_{H,i} \tilde{m}_{H,i+1} + \tilde{m}_{H,i} \bar{m}_{n,i+1})$;
        // Compute remaining belief mass after adding node
           $i + 1$ to aggregated beliefs
    end
    // After the final iteration, for $n = 1, \ldots, l$, $c_{n,|\beta|}$
       represents the combined belief masses for all factors
       in $\beta$.
    for $n = 1 \rightarrow l$ do
        $\beta_n \leftarrow \frac{c_{n,|\beta|}}{1 - \bar{c}_{H,|\beta|}}$;
    end
    $\beta_H \leftarrow \frac{\tilde{c}_{H,|\beta|}}{1 - \bar{c}_{H,|\beta|}}$;
    // Compute the aggregated Belief function $\beta$ (and the
       ignorance-level $B_H$) for $\beta$
    return $\beta$
end
```

Algorithm 2: `aggregateBeliefs` function.