Believing Software: A Method of Practical Proof for Software Engineering

Jerry Overton
Computer Sciences Corporation (CSC)
St. Louis, Missouri, USA
joverton@csc.com

Abstract – **For years, software engineers have tried to achieve the same collective confidence in their software specifications that mathematicians, by way of proof, have in their theorems. Most attempts have been rooted in deduction and have produced methods that are too difficult to use in practice. By borrowing from mathematics its methods of recording, communicating, and scrutinizing arguments instead of its methods of deduction, we introduce a method practical proof in software engineering. The result of this work is a cost-effective method for getting consensus among practicing software engineers about the adequacy of a real-world software design.**

Keywords – Consensus, Proof, Software Engineering, Software Design Pattern, Practical Formal Method, POAD Theory.

I. INTRODUCTION

This paper is an elaboration of the ideas originally published in [1]. We expand on the method of practical mathematical reasoning in software engineering; provide a more detailed account of how the method can be used to argue for the adequacy of a real-world software system; and provide an extended analysis of the significance of this research.

One of the most distinguishing features of mathematics is the level of consensus among mathematicians about the truth or falsehood of their theorems [2]. Mathematicians, by way of proof, enjoy an unusually high collective confidence in their theorems. For years, software engineers have tried to achieve the same collective confidence in their software specifications [3]. So far, most attempts have been limited to verifying software using some form of deduction [4] – an approach rooted in the assumption that proof happens as a result of deductive calculation [2]. Deductive methods all have the same drawback: the cost (in time and effort) of using them to verify a software design is usually an order of magnitude greater than the cost of creating the design itself [5]. Deductive methods of verification are so expensive that, in practice, they are used only to reduce the risk of the most serious design flaws – flaws that may compromise human safety, for example [6].

In this work, we borrow proof from mathematics; use it to argue for the fitness-for-purpose of a software design; and do so in an amount of time that is within same order of magnitude that it took to create the design itself. But rather than assuming that proof is achieved through a series of deductive calculations, we adopt, instead, the view that proof is achieved by a gradual process of collective scrutiny and refinement [3]:

> *First of all, the proof of a theorem is a message. A proof is not a beautiful abstract object with an independent existence. No mathematician grasps a proof, sits back, and sighs happily at the knowledge that he can now be certain of the truth of his theorem. He runs out into the hall and looks for someone to listen to it. He bursts into a colleague's office and commandeers the blackboard. He throws aside his scheduled topic and regales a seminar with his new idea. … If the various proofs feel right and the results are examined from enough angles, then the truth of the theorem is eventually considered to be established.*

We borrow from math its methods of recording, communicating, and scrutinizing arguments – not its methods of deduction. First, we use Pattern-Oriented Analysis and Design (POAD) Theory [7], [8] to structure an adequacy argument based on software design patterns (the details of POAD Theory are given in Section IV, subsection B). Then, we use fuzzy inference to argue that the particular pattern instantiations in the design makes it fit for purpose. The result is what we will refer to as practical proof in software engineering: a cost-effective method for getting consensus among practicing software engineers about the adequacy of a real-world software design.

The rest of this paper is laid out as follows. We start by placing this work within the wider context of existing research on software design verification. Next, we specify the design for a collaborative wireless sensor network – the real-world problem of interest. We use POAD Theory to structure a proof-of-correctness argument for the design and calculation (based on fuzzy inference) to complete the argument. Finally, we close with an analysis of this work and conclusions about its significance.

II. STATE OF THE ART

The prior art for this research is the body of existing proof-of-correctness methods for computer programs. In software engineering, requirements are specifications proposed in the requirements phase and design is the specification proposed in the design phase. Proof-of-correctness happens when it is demonstrated that a design meets its requirements. Proof-of-correctness techniques reduce to a step-by-step reasoning for determining whether or not the design is fit for purpose [9]. Requirements dictate acceptable systems behavior by defining a mapping between

a set of pre-states and a set of post- states [10]. To satisfy a set of requirements, a design must take as input each pre-state and produce as output the prescribed post-state. Regardless of the specific technique, proof-of-correctness happens by process of refinement; where the original specifications of the requirements are replaced by the equivalent or stronger specifications of the design [10]. The body of existing proof-of-correctness methods is vast; however, they all work according to one of the three fundamental laws of refinement: refinement by steps, refinement by parts, and refinement by cases [10].

In refinement by steps, proof-of-correctness proceeds by sequential actions where, in each step, a part of the requirement specification is replaced by a suitable design. The refinement continues until all requirements have been interpreted as sequences of computational steps. In practice, proof-of-correctness techniques based on refinement by steps work by using semantic rules for interpreting requirements, specifying designs, and making comparisons between the two. For example, [11], [12], and [13] all develop competing formal semantics that makes it possible to prove (by steps) the correctness of designs documented in UML state chart diagrams.

In refinement by parts, an analyst normalizes the requirements into orthogonal parts, and then independently replaces each part with a suitable design element. In practice, proof-of-correctness using refinement by parts proceeds by normalizing requirement specifications into domains [14]. The requirements of each domain are replaced by designs represented by mathematical constructs – for example, partial functions as in [15]; actor-based models as in [16]; or by games between the environment and the system as in [17].

In refinement by cases, requirements are specified in terms of a correspondence between pre and post conditions. In Hoare Logic [18], for example, the central construct is the Hoare Triple that relates a pre-condition to a post-condition by way of a command. Refinement occurs by replacing a requirement with a design that achieves the same correspondence.

Existing proof-of-correctness methods (whether they use refinement by steps, parts, or case) require that requirements be replaced by suitable designs and that those replacements be justified by deductive implication [10]. As mentioned in the Introduction, deduction is expensive to use in the proof-of-correctness of real software systems – about an order of magnitude more expense that the cost of creating the design itself. Lightweight formal methods [5] are a way of compensating for the high cost; but instead of reducing the cost of deduction, lightweight formal methods simply limit its use. The central problem in the current state of the art remains – current methods of proof-of-correctness are too expensive for general use in real-world systems.

This research breaks from the state of the art by rejecting the restriction that deduction must be used to justify refinement. Instead, we will propose a proof-of-correctness technique based on Problem Oriented Software Engineering (POSE) [19] and software design patterns. The details of POSE are given in Section IV, subsection A. Deduction is one of many methods for justifying the substitution of requirements with engineering designs. We do not evaluate our method of justification by comparing it to deduction. Instead, in Section V, we evaluate our method of justification by determining whether or not it is logically sound (a standard more general than deduction).

POSE provides a framework for accepting engineering expertise as justification for replacing a requirement with a design. We complement POSE by using software design patterns as ready-made units of justification and engineering expertise. There are prior works that combine both formal methods and software design patterns. Most of these works (for example [20], [21], and [22]) offer proposals for formally representing software design patterns, but they do not offer methods for proof-of-correctness. The works that do offer proof-of-correctness methods (such as [23], [24], and [25]) do so based on deductive calculation; and, therefore, have the same drawbacks as the rest of the works surveyed.

POSE gives us the freedom to choose a more efficient method of reasoning. Software design patterns allow us to easily connect our arguments to the processes of collective scrutiny and feedback already in existence in the pattern community [26]. In the course of this research, we contribute to the state of the art a proof-of-correctness technique that is closer to real-world use of proof in mathematics [3]: rigorous arguments (but not deductive arguments) whose truth is established by a social process of scrutiny and feedback; arguments whose truth could be demonstrated by formal deduction if it were worth the time and effort.

III. A COLLABORATIVE SYSTEM DESIGN

In this section we introduce a real-world design for a software system. This design will be the target of analysis and proof-of-correctness in subsequent sections.

In collaborative systems, otherwise autonomous computing nodes cooperate to achieve a common task that would not be possible with any individual node acting alone [27]. Although the exact definition of a collaborative system can vary depending on context, in this paper, we focus on three defining characteristics:

- Nodes in collaborative systems are autonomous and spatially distributed.
- Task-execution responsibilities are distributed across multiple nodes.
- The communication links between nodes are decentralized and dynamic.

Figure 1 is an example of a collaborative system – a network of environmental sensor stations. The system is designed to report the environmental condition of a given

geographic region. Each sensor is capable of recording and reporting its local conditions, but to record and report the condition of the entire region requires all sensor stations to cooperate.

The nodes in the network are autonomous and spatially distributed across the region shown. Each sensor is capable of recording and reporting its local environmental conditions without the help of any of the other sensor stations. Task-execution is distributed across multiple nodes since reporting conditions for the entire region requires the cooperation of multiple sensor stations. The communication links between the sensor stations are decentralized and dynamic. Sensors can enter and leave the network at anytime. Every station is wirelessly connected to every other station, so no single sensor failure can disrupt the overall network connectivity.

Figure 1: Example Collaborative System [28].

In our system, we expect that node failures will be common and that the wireless communication links will be prone to frequent interruptions. For example, the sensor stations are exposed to adverse weather, they are knocked over and broken easily, and they can be expected to run out of power. People, cars, and animals passing between two sensor stations can cause a temporary loss of communication between them. If any of these things happen at the right time, a controller in a region may miss a sensor update and become out of touch with the current conditions in the region.

A robust design will allow the sensor stations (referred to from now on as nodes) to both detect and mitigate these kinds of failures. Each node must be designed to detect when other nodes become unresponsive; each node must be designed to perform in degraded mode when disconnected from the network; and the network must be capable of using node redundancy to compensate for the loss of any particular node. A satisfactory design must satisfy the following requirements.

Req. 1. *Group Communication*. Each node must be able to communicate with all other nodes and detect when a node becomes unresponsive.

Req. 2. *Fault Tolerance*. The network must be capable of using node redundancy to compensate for the loss of any particular node.

Req. 3. *Degraded Mode Operation.* Each node must be capable of performing limited functions while disconnected from the network, and be capable of resuming full function when network communication is restored.

Figure 2 shows the class diagram of our design for a robust collaborative system. We consider **Figure 2** to be the class diagram of a real-world design since it was taken from the design of an actual software system built to provide fault tolerance in collaborative systems [29]. Each *GroupNode* operates in its own thread of execution. Each node gets its ability to collaborate through an association with a *CommStrategy* object. The *CommStrategy* has an association back to its *GroupNode* in case the *GroupNode* needs to be notified of events from the *CommStrategy*. The *PushPullNode* (which, represents a sensor or controller) is a specific type of *GroupNode*. The *PushPullStrategy* is a specific type of *CommStrategy*. Using the JGroup communication API [30] the *PushPullStrategy* gives each *PushPullNode* the ability to communicate with other *PushPullNodes*.

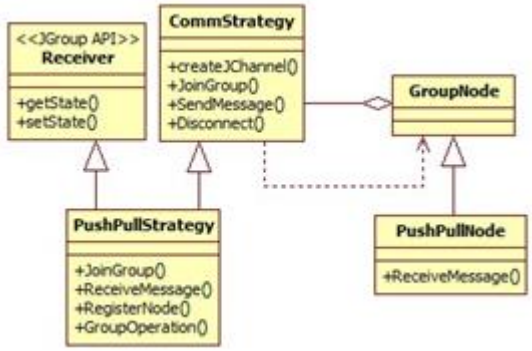

Figure 2: Class diagram of a design for a robust collaborative system.

Figure 3 is a sequence diagram of how nodes participate in group operations. Sensors A and B are controlled by the Controller. Sensors A and B join the same group representing a single physical zone. The Controller relies on both sensor A and sensor B to report temperature for a given region. The controller doesn't care which sensor it uses as long as at least one of them is always available. When the Controller wants a temperature reading from the zone, it joins the zone's group and executes *CommStrategy.groupOperation()*. JGroups elects a leader within the group and calls *getState()* on that node (let's assume that sensor A was chosen). The *getState()* operation of sensor A takes a temperature reading and sets the reading as the operation's return value. JGroups then calls *setState()* on the Controller, passing it the temperature reading from sensor A. In subsequent requests for the zone temperature, if sensor A becomes unresponsive, JGroups will failover to sensor B.

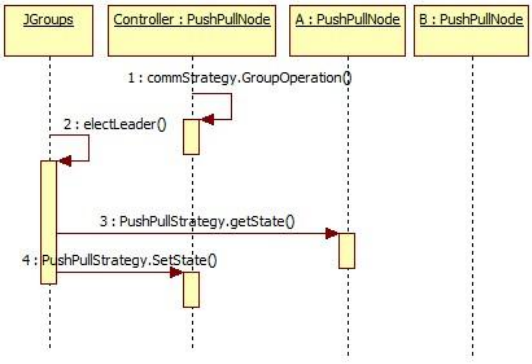

Figure 3: Nodes participating in a group operation

We have a design, but is it a good one? Does it solve our problem and satisfy our requirements? In the remainder of this paper, we will construct a proof-of-correctness-argument for the design.

IV. OUR METHOD FOR PROOF OF CORRECTNESS

A. Our Approach: The Basis

The method that we use in the next section to structure our proof-of-correctness argument (POAD Theory) is based on a system of reasoning known as Problem-Oriented Software Engineering (POSE) [19]. In POSE a software engineering problem has context (a real-world environment), W; a requirement, R; and a solution (which, may or may not be known), S. We write $W, S \vdash R$ to indicate that we intend to find a solution S that, given a context of W, satisfies R. Details about an element of the problem can be captured in a description for that element; and a description can be written in any language (UML in our case) considered appropriate. The problem, P_0 , of designing a collaborative system can be expressed in POSE as:

$$CSystem: W, S \vdash R \qquad (1)$$

where W is the real-world environment for the system (shown in **Figure 1**); S is the system itself and R are requirements Req. 1, Req. 2, and Req. 3. Equation (1) says that we can expect to satisfy R when the system S is applied in context W.

In POSE, engineering design is represented using a series of problem transformations. Transformation steps can be arbitrary in size; large steps can be composed of smaller ones. A problem transformation is a rule where a conclusion problem $P: W, S \vdash R$ is transformed into premise problems $P_i: W_i, S_i \vdash R_i, i = 1, \dots, n\ (n > 0)$ using justification J and a rule named N, resulting in the transformation step $\frac{P_1 \dots P_n}{P}\ \frac{[N]}{\ll J \gg}$. This means that S is a solution of $W, S \vdash R$ whenever $S_i, \dots, S_n$ are solutions of $W_i, S_i \vdash R_i, \dots, W_n, S_n \vdash R_n$. The justification J collects the evidence of adequacy of the transformation step and is validated by all relevant stake-holders. Through the application of rule N_i, problems are transformed into other problems that may be easier to solve or that may lead to other problems that are easier to solve. These transformations occur until we are left only with problems that we know have a solution fit for the intended purpose. POSE allows us to use one big-step transformation to represent several smaller ones. We can apply big-step transformations without having completed justification, with the understanding that we will complete the justification later and solve our problem. The progression of a software engineering solution described by a series of transformations can be shown using a development tree.

$$\frac{\dfrac{\overline{P_3: W_3, S_3 \vdash R_3} \quad P_4: W_4, S_4 \vdash R_4 \quad [N_2]}{P_2: W_2, S_2 \vdash R_2 \ll J_2 \gg} \quad [N_1]}{P_1: W_1, S_1 \vdash R_1 \ll J_1 \gg} \qquad (2)$$

In the tree, the initial problem forms the root and problem transformations extend the tree upward toward the leaves. There are four problem nodes in the tree: P_1, P_2, P_3, and P_4. The problem transformation from P_1 to P_2 is justified by J_1; the transformation from P_2 to P_3 and P_4 is justified by J_2. The bar over P_3 indicates that P_3 is solved. Because P_4remains unsolved, the adequacy argument for the tree (the conjunction of all justifications) is not complete, and the problem P_1 remains unsolved. A complete and fully-justified problem tree means that all leaf problems (in this case P_3 and P_4) have been solved.

For the sake of clarity, we will show the context, solution, and requirement of a problem only when necessary to understanding a given transformation. In many of the subsequent equations, these details are omitted and only the problem's name is shown.

B. Our Approach: Practical Mathematics

In this section, we introduce POAD theory and use it to structure the argument that the design from Section III is fit-for-purpose.

A software design pattern is a tool that a software engineer can use to take a complex, unfamiliar problem and transform it into simpler, more familiar ones [31]. The basis of POAD Theory is that software engineering design can be represented as a series of transformations from complex engineering problems to simpler ones, and software design patterns can be used to justify those transformations:

$$\frac{SimplerProblem}{ComplexProblem}\ \frac{[SolInt]}{\ll Pattern_1, \dots, Pattern_n \gg} \qquad (3)$$

In (3) the patterns $Pattern_1, \dots, Pattern_n$ are used to justify the transformation from the $ComplexProblem$ to the

$SimplerProblem$. The engineering expertise documented in the Object Group pattern describes how to achieve reliable multicast communication among objects in a group [32]. The pattern gives us justification for substituting $CSystem$ with the easier problems of implementing a communication mechanism ($Comm$) and implementing an object that uses the communication mechanism (Obj). We write this as

$$\frac{Comm\ Obj}{CSystem: W, S \vdash R} [SolInt] \ll OG \gg \quad (4)$$

which, means that we used the engineering expertise in the Object Group pattern (represented as $\ll OG \gg$) to justify a solution interpretation (represented by the rule [$SolInt$]) from $CSystem$ to $Comm$ and Obj.

But there is a problem. Equation (4) implies that if we have a solution to $Comm$ and Obj then we also have a solution to $CSystem$. Having a communication mechanism that allows for reliable multicast communication and objects capable of communicating that way may be sufficient to argue that the solution can satisfy the group communication (Req. 1) and fault tolerance requirements (Req. 2); but the solution does not address the requirement that the objects be capable of degraded mode operation (Req. 3).

We can add to our solution as many transformations as necessary. We can add to (4), a transformation justified by the Explicit Interface pattern [33].

$$\frac{Comm\ \frac{Intf\ Node}{Obj} [SolInt] \ll EI \gg}{CSystem: W, S \vdash R} [SolInt] \ll OG \gg \quad (5)$$

The Explicit Interface pattern describes how to achieve separation between an object and its environment [33]. We can use that separation to argue that the nodes in our design can function even when disconnected from each other.

Equation (5) is a solution tree with $CSystem$ at the root. Two problem transformations extend the tree upward toward the leaves $Comm$, $Intf$, and $Node$. The equation structures an argument whose adequacy is established by the conjunction of all justifications – in this case by the engineering expertise contained in the Object Group pattern $\ll OG \gg$ and the engineering expertise contained in the explicit interface pattern $\ll EI \gg$. A solved problem is written with a bar over it; for example, if the Object Group pattern were sufficient to convince us that we have an adequate communication mechanism, then we could rewrite (5) as follows

$$\frac{\overline{Comm}\ \frac{Intf\ Node}{Obj} [SolInt] \ll EI \gg}{CSystem: W, S \vdash R} [SolInt] \ll OG \gg \quad (6)$$

where the bar over $\overline{Comm}$ indicates that we have sufficient justification to consider that problem solved. A complete and fully-justified problem tree means that all leaf problems – for (5), the leaves are $Comm$, $Intf$, and $Node$ – have been solved. We complete the problem tree in (5) by adding transformations and justifications sufficient to solve all leaf problems.

$$\frac{\overline{Recvr}}{Comm} [SolInt] \ll J_1 \gg \quad \frac{\overline{PPStrat}}{Intf} [SolInt] \ll J_2 \gg \quad \frac{\overline{PPNode}}{Node} [SolInt] \ll J_3 \gg \quad (7)$$

Equation (7) continues the solution from (5) by providing solutions for all leaf problems in (5). In (7) the problems $Recvr$, $PPStrat$, and $PPNode$ correspond to the *Receiver*, *PushPullStrategy* and *PushPullNode* (from **Figure 2**) respectively. Each leaf problem from (7) is a design implementation of the patterns chosen in (5). The $Recvr$is an implementation of the communication mechanism prescribed by the Object Group pattern, $PPStrat$ is an implementation of the interface prescribed by the Explicit Interface pattern, and $PPNode$ is an implementation of the domain object prescribed by the Explicit Interface pattern. By considering (5) in combination (7), we can conclude that, given sufficient justification (J_1, J_2, and J_3), we can consider our original problem ($CSystem$ from (5)) solved. In other words, once we find J_1, J_2, and J_3, we will have a complete proof-of-correctness argument for the design described in Section III.

C. Our Approach: Practical Calculations

So far, we have a general argument for how to use software design patterns to solve our problem, but it isn't clear how this general argument relates to our specific design. In this section, we introduce a method of calculation – based on Fuzzy Inference [34] – that connects our more general argument to the specific design decisions represented by the *Receiver*, *PushPullStrategy* and *PushPullNode* elements of **Figure 2**. We use the calculation results as the justification (J_1, J_2, and J_3) needed to complete the proof-of-correctness argument for (5) and (7).

Fuzzy inference is based on a generalized modus ponens [34] where arguments take the form:

$$\begin{array}{c} If\ A\ Then\ B \\ A' \\ Therefore\ B' \end{array} \quad (8)$$

For example, suppose we accepted the general rule that: *if the Object Group pattern were well implemented as part of our collaborative system, then the group communication of*

our system would be good. If we knew that, in our system, the Object Group pattern were implemented poorly, then fuzzy inference would allow us to conclude that the group communication of the system would also be poor.

We apply fuzzy inference to statements about the use of software design patterns to create a technique for calculating the results of software design decisions. In works such as [35], [36], and [37] fuzzy logic has been used in combination with design patterns to reverse engineer a design from source code. A software design pattern can have several different implementations. These works use fuzzy inference to determine if an existing solution, known to satisfy certain requirements, matches a general design pattern. We apply this same idea, but in reverse: for a given design pattern, we use fuzzy inference to determine if a particular implementation of that pattern will lead to a solution that we can trust will satisfy particular requirements. For all fuzzy logic operations (such as creating fuzzy input variables, performing fuzzy inference, and visualizing fuzzy output variables), we used Mathematica's Fuzzy Logic Environment [38].

We begin our calculation by creating fuzzy rules [34] that represent the design constraints introduced by (5) and (7):

Rule 1. If the object group pattern is implemented then group communication will be good

Rule 2. If the object group pattern is not implemented then fault tolerance will be low

Rule 3. If the explicit interface pattern is not implemented then degraded-mode operation will not be enabled

Rule 4. If the push pull node communicates statically then degraded-mode operation will not be enabled

Rule 5. If the push pull node communicates dynamically and the explicit interface pattern is implemented then degraded-mode operation will be enabled

Rule 6. If the push pull node communicates dynamically and the object group pattern is implemented then group communication will be good and fault tolerance will be high.

Each rule makes statements concerning input and output variables. Each variable has membership functions [34] that allow the inference engine to turn the numeric values of the variables into the more intuitive concepts used in Rules 1-6. For example, **Figure 4** shows the three membership functions (Poor, Good, and Moderate) for the Group Communication output variable. From the shape of the membership functions, we can see that a Group Communication variable with a value of 0.7 would be considered mostly moderate, slightly good, and not at all poor.

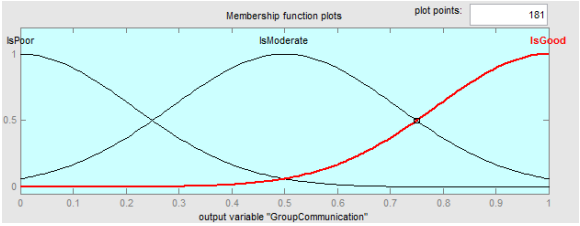

Figure 4: Membership function for the Group Communication output variable

The fuzzy rules capture our understanding of how the software engineering expertise contained in $\ll OG \gg$ and $\ll EI \gg$ (from (5)) relates to the original requirements R of (1). In our calculation, $Recvr$, $PPStrat$, and $PPNode$ (from (7)) are represented using input fuzzy variables and Req. 1, Req. 2, and Req. 3 (from (1)) are represented using output fuzzy variables. As shown in **Figure 5**, input variables representing our implementation choices are fed into an inference engine which, has been loaded with Rules 1-6. The inference engine produces values for the fuzzy output variables which, represent the results of our calculation.

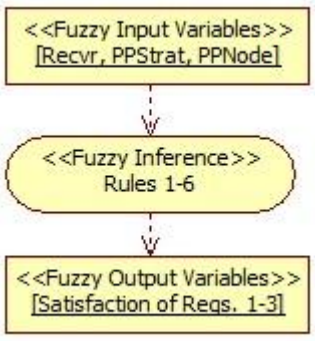

Figure 5: Process flow of the simulation.

We calculate the design choices made in (7) by assigning specific values to the fuzzy input variables $Recvr$, $PPStrat$, and $PPNode$. Because JGroups provides a faithful implementation of the Object Group pattern, the $Recvr$ provides an almost-complete implementation (0.949) of the Object Group pattern's communication mechanisms. The $PPNode$ is a reasonably good approximation (0.762) of the Object Group's node element; but the communication strategy provided by the $PPStrat$ is not a very good representation (0.584) of intent of the Explicit Interface pattern. The $PPStrat$ object separates from $PPNode$ the details of group communication, but, unlike a true explicit interface, still requires $PPNode$ to select an appropriate instance of $PPStrat$ based on the current circumstances.

The results of the calculation predict that the design decisions described in (7) will result in a collaborative system that satisfies Req. 1-3 (see **Figure 6**). The results of the calculation are that the system will have good group communication (0.833), good fault tolerance (0.815), and will operate well in degraded mode (0.807).

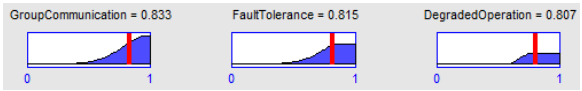

Figure 6: The results of collaborative system calculation.

If we compare every possible design choice to its corresponding calculated result, we get a design space that shows how design choices affect the quality of the system. **Figure 7** shows the design space for achieving the desired fault tolerance. There are a number of design choices for the *Recvr* and *PPNode* elements (shaded in yellow) that will result in acceptable (0.7 or greater) fault tolerance for the system. The design space shows that fault tolerance is most dramatically affected (indicated by the surface's steep drop-off) by the design of the *Recvr* – which, makes sense because that portion of the design determines the group communication capabilities.

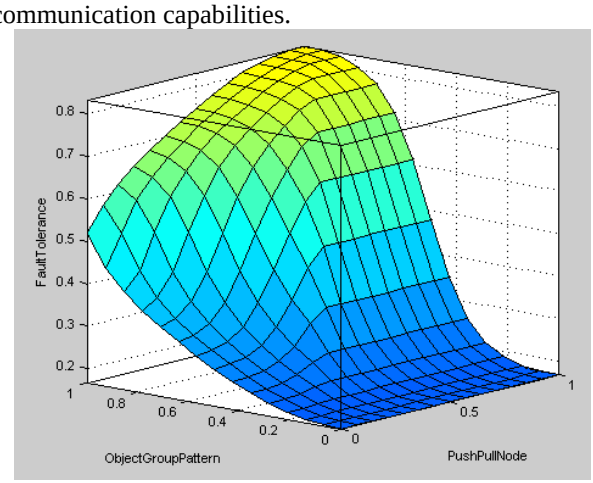

Figure 7: Fault tolerance design space

Figure 8 shows the design space for achieving the desired degraded-mode operation for each node in the collaborative system. The number of acceptable design choices for the *PPStrat* and *PPNode* are more limited than the choices available in the design space of **Figure 7**. The choice of design for *PPNode* seems to be slightly more influential to degraded-mode operation than the design choices for *PPStrat*.

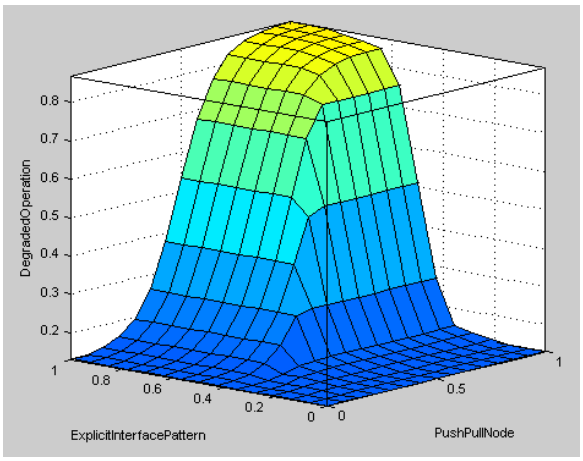

Figure 8: Degraded-mode operation design space

The positive calculation results (which, are also confirmed by our analysis of the design spaces) provides the justification (J_1, J_2, and J_3), needed to complete the proof-of-correctness argument for the design of **Figure 2**.

The argument is complete, but is it trustworthy? Can we expect the method of argument described here to be sufficient to build consensus among practicing software engineers that our design meets its requirements? In the remainder of this paper, we perform a critical analysis with the goal of answering these questions.

V. ANALYSIS AND CONCLUSIONS

We have proposed a method of proof-of-correctness for software design. Keep in mind that by proof-of-correctness, we mean some method for convincing our audience that a design meets its requirements. We started with the real-world problem of designing a collaborative system. We used POAD Theory to create a general argument; we used software design patterns to justify the argument; and we used calculation to apply the general argument to our specific design. Our goal was to introduce a cost-effective method for getting consensus among practicing software engineers. We analyze whether or not our method accomplishes our goal by considering the following questions: is our proposed method trustworthy, is it convincing, is it practical?

Is our method trustworthy? We can consider our method trustworthy if we can show that it is sound: given premises that can be trusted, our method will produce conclusions that can be trusted. Our method consists of a general argument based on POAD Theory and specific calculations based on Fuzzy Logic. POSE transformations – the basis of POAD Theory – are sound. Premise problems can only be interpreted as conclusion problems given sufficient justification for doing so. The original problem is considered solved only after all leaf-level sub-problems are known to be solved. In POAD Theory a solved problem is made only of known-solved sub-problems; and the break-

down of problems into constituent sub-problems is fully-justified. Generalized modus ponens – the basis of fuzzy inference – is also sound in that its conclusions are true if the premises are true [39]. We can trust the results of our calculation as long as we trust the rules that we establish for governing the simulation.

Is our method convincing? Whether or not the particular argument given by (5), (7) and the justification from Section IV, Subsection C is convincing will depend on the results of a social process among practicing software engineers. The argument will have to generate interest and credibility among some initial group of engineers. It will have to be circulated among a wider audience, polished and refined. A truly convincing argument will be internalized by engineers. That is, practicing software engineers may attempt to use parts or all of the argument to justify designs of their own; or the design itself will be routinely copied and used in other working IT software systems. We can, however, determine if our general proof-of-correctness method is capable of producing convincing arguments. We can compare the methods described here to the method of proof used in mathematics – the social process of scrutinizing humanly understandable (as opposed to purely formal) arguments [40]. Therefore, we focus the analysis of whether or not our method is convincing by asking, instead: *does our method encourage the creation and collective scrutiny of understandable arguments?*

Our method is, essentially, an application of analogical reasoning – one of the basic patterns of human reasoning [41]. Our method makes arguments understandable by replacing the more difficult task of predicting the consequences of a design with the much easier task of comparing a design with known software design patterns. The calculations of Section IV, Subsection C draw a comparison between the design of Section III and the interaction of software design patterns given by (5) and (7). We reason that the closer our design is to the solutions described in the design patterns, the closer our results will be to the consequences described in the design patterns. Our calculation tells us just how close our design needs to be in order to produce satisfying results. POAD Theory allows us to record the argument so that it can be read, circulated, and scrutinized (as evidenced by this publication). Further, using software design patterns, we build on the processes of collective scrutiny and feedback already in existence in the pattern community [26].

Is our method practical? With a relatively small amount of effort (roughly the same amount of time it took to create the original design), we were able to use math and calculation to discover things about the design that are not obvious. With Eq. 1-7 and the associated explanatory text, we were able to create a mathematical model that had a meaningful correspondence to the collaborative system design in Section III. We were able to use those equations to structure an argument for the design's adequacy and to predict that: given the argument structure defined by (5) and (7); and the engineering expertise contained in the Object Group and Explicit Interface patterns; all we needed to validate the design of Section III was to find justifications J_1, J_2, and J_3. Using Rules 1-6; fuzzy variable membership function definitions (the membership function for the Group Communication output variable is shown in **Figure 4**); and fuzzy inference; we were able to simulate the effect that the design choices of (7) would have on the resulting system qualities (shown in **Figure 6**).

Although we are able to argue that the method we describe here is capable of practical proof-of-correctness, we consider it to be a greater accomplishment to demonstrate that particular arguments based on this method are convincing. That is, our ultimate goal is to produce arguments that are trusted enough to become the infrastructure for a particular field of endeavor in software engineering. We recognize that gaining consensus and confidence in an argument will likely requirement more than just argument creation and discussion. We recognize the need to empirically demonstrate the ability of our proposed methods. At CSC, we are currently using this method to predict and manage risk in large-scale data center migration. Effectively managing risk for such large-scale endeavors requires high levels of consensus and coordination among migration teams. The methods described in this research are being used to identify ideas that are most likely to result in a better understanding and mitigation of the risk factors involved. We are exploring whether or not our methods are effective in identifying a set of ideas that a community of CSC engineers can rely on to improve performance in some of our most complex projects.

ACKNOWLEDGMENT

We would like to thank Dariusz W. Kaminski of the Marine Scotland directorate of the Scottish Government for his insightful review and commentary.

REFERENCES

[1] J. Overton. *Practical Math and Simulation in Software Design.* Proceedings of the Third International Conferences on Pervasive Patterns and Applications (Computation World 2011). 2011.

[2] R. Hersh. *What is Mathematics, Really?* Oxford University Press, New York, Oxford, 1997.

[3] R. De Millo, R. Lipton, and A. Perlis. *Social Processes and Proofs of Theorems and Programs*, Communications of the ACM, Volume 22, Number 5, 1979.

[4] G. Holzmann. *Trends in Software Verification.* Proceedings of the Formal Methods Europe Conference (FME'03). 2003.

[5] D. Jackson. *Lightweight Formal Methods.* FME 2001: Formal Methods for Increasing Software Productivity, Lecture Notes in Computer Science, Volume 2021, 2001

[6] J.P Bowen. *Formal Methods in Safety-Critical Standards*. In Proceedings of 1993 Software Engineering Standards Symposium

(SESS'93), Brighton, UK, IEEE Computer Society Press, pages 168-177, 1993.

[7] J. Overton, J. Hall, L. Rapanotti, and Y. Yu. *Towards a Problem Oriented Engineering Theory of Pattern-Oriented Analysis and Design*. In Proceedings of 3rd IEEE International Workshop on Quality Oriented Reuse of Software (QUORS), 2009.

[8] J. Overton, J. G Hall, and L. Rapanotti. *A Problem-Oriented Theory of Pattern-Oriented Analysis and Design.* 2009, Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, pages 208-213, 2009.

[9] W. Adrion, M. Branstad, and J. Cherniavsky. *Validation, Verification, and Testing of Computer Software.* ACM Computing Surveys, Vol. 14, No.2, pages 159-192, June 1982.

[10] E. Hehner. *A Practical Theory of Programming.* Springer-Verlag, New York, 1993.

[11] D. Latella, I. Majzik and M. Massink. *Towards A Formal Operational Semantics of UML Statechart Diagrams.* Third International Conference on Formal Methods for Open Object-Oriented Distributed Systems, pages 331-347, Kluwer Academic Publishers, 1999.

[12] D. Alexandre, M. Moller, and W. Yi. *Formal Verification of UML Statecharts with Real-time Extensions*. Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002, volume 2306 of LNCS, pages 218–232. Springer–Verlag, 2002.

[13] G. Kwon. *Rewrite Rules and Operational Semantics for Model Checking UML Statecharts*. Proceedings of the 3rd International Conference on UML, Lecture Notes Comp. Sci. 1939, pages 528–540, 2000

[14] D. Scott. *Domains for Denotational Semantics.* In Proceedings of ICALP, 1982.

[15] R. Keller. *Formal Verification of Parallel Programs.* Communications of the ACM Volume 19, No. 7 pages 371-384. 1976

[16] M. Sirjani, A. Movaghar, A. Shali, and F. de Boer. *Modeling and Verification of Reactive Systems using Rebeca*. Fundamenta Informaticae, pages 385–410, 2004.

[17] S. Abramsky, D. R. Ghica, A. S. Murawski, and C.-H. L. Ong. *Applying Game Semantics to Compositional Software Modeling and Verification*. In TACAS'04, volume 2988 of Lecture Notes in Computer Science, pages 421–435, 2004.

[18] C. Hoare. *An Axiomatic Basis for Computer Programming.* Communications of the ACM, Volume 12, No. 10, pages 576-583, 1969.

[19] J. G. Hall, L. Rapanotti, and M. Jackson. *Problem-Oriented Software Engineering: Solving the Package Router Control Problem.* IEEE Trans. Software Eng., 2008. doi:10.1109/TSE.2007.70769

[20] T. Taibi and D. Ngo. *Formal Specification of Design Pattern Combination Using BPSL*, Information and Software Technology 45, Elsevier, pages 157–170, 2002.

[21] N. Soundarajan and J. Hallstrom. *Responsibilities and Rewards: Specifying Design Patterns*, Proceedings of the 26th International Conference on Software Engineering (ICSE'04), pages 666-675, May 2004.

[22] P. Alencar, D. D. Cowan, and C. J. P. Lucena. *A Formal Approach to Architectural Design Patterns*, Proceedings of the 3rd International Symposium of Formal Methods Europe, pages. 576-594, 1995.

[23] D. J. Ram, P. J. K. Reddy, and M. S. Rajasree. *An Approach to Estimate Design Attributes of Interacting Patterns.* http://dos.iitm.ac.in/djwebsite/LabPapers/JithendraQAOOSE2003.pdf, Last Accessed: 30 January 2011.

[24] J. Paakki, A. Karhinen, J. Gustafsson, L. Nenonen, and A. Verkamo. *Software Metrics by Architectural Pattern Mining.* In Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress), peges 325–332, 2000.

[25] P. Tonella and G. Antoniol. *Object Oriented Design Pattern Inference.* In Proceedings of the IEEE International Conference on Software Maintenance. IEEE Computer Society Washington, DC, USA, 1999.

[26] N. Harrison. *The Language of Shepherds.* http://hillside.net/plop/plop99/proceedings/harrison/shepherding4.pdf, Last Accessed: 06/21/2012.

[27] T. Clouqueur, K.K. Saluja, and P. Ramanathan. *Fault Tolerance in Collaborative Sensor Networks for Target Detection*. IEEE Transactions on Computers. Vol. 53, No. 3, pages 320-333, March 2004.

[28] http://www.citysense.net, Last Accessed: 1/27/2012

[29] J. Overton. *Collaborative Fault Tolerance using JGroups*. Object Computing Inc. Java News Brief, 2007, http://jnb.ociweb.com/jnb/jnbSep2007.html, Last Accessed 02/05/2012.

[30] The JGroups Project. http://www.jgroups.org/. Last Accessed 06/21/2012

[31] F. Buschmann, K. Henney, and D. Schmidt. *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*, Volume 5. John Wiley & Sons, West Sussex, England, 2007.

[32] S. Maffeis. *The Object Group Design Pattern.* In Proceedings of the 1996 USENIX Conference on Object-Oriented Technologies, (Toronto, Canada), USENIX, June 1996.

[33] F. Buschmann, K. Henney, and D. Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing (Wiley Software Patterns Series)*, Volume 4. John Wiley & Sons, 2007.

[34] K. Tanaka. *An introduction to Fuzzy Logic for Practical Application.* Berlin: Springer, 1996.

[35] J. Niere. *Fuzzy Logic Based Interactive Recovery of Software Design*. Proceedings of the 24th International Conference on Software Engineering, Orlando, Florida, USA, pages 727-728, 2002.

[36] C. De Roover, J. Brichau, and T. D'Hondt. *Combining Fuzzy Logic and Behavioral Similarity for Non-strict Program Validation.* In Proceedings of the 8th Symposium on Principles and Practice of Declarative Programming, pages 15–26, 2006.

[37] I. Philippow, D. Streitferdt, M. Riebisch, and S. Naumann. *An Approach for Reverse Engineering of Design Patterns.* Software Systems Modeling, pages 55–70, 2005.

[38] http://www.wolfram.com/products/applications/fuzzylogic/, Last Accessed: 06/23/2012

[39] S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Englewood Cliffs, NJ: Prentice Hall, 1995.

[40] W. Thurston. *On Proof and Progress in Mathematics.* Bulletin of the American Mathematical Society. Volume 30, pages 161-177, 1994.

[41] G. Polya. *Mathematics and Plausible Reasoning: Volume II, Patterns of Plausible Inference.* Princeton University Press. 1968.