# Practical Math and Simulation in Software Design

Jerry Overton
Computer Sciences Corporation (CSC)
joverton@csc.com

**Abstract – Formal verification of a software design is often much more costly than producing the design itself; and formal methods usually have limited use in real-world software design. In this work, we propose cost-effective methods – based on software design patterns – of applying mathematics and simulation to real-world software designs.**

*Keywords – Software Design Pattern, Practical Formal Method, POAD Theory*

## I. INTRODUCTION

The cost of using formal methods to verify a software design is usually an order of magnitude greater than the cost of creating the design itself [1]. For many projects, formal methods are only worth using for reducing the risk of the most serious errors – flaws that may affect safety, for example [2]. We propose practical (cost-effective) methods of mathematics and simulation for real-world software designs. We introduce a method for validating the adequacy of software designs based on the mathematics of Pattern-Oriented Analysis and Design (POAD) Theory [3], [4] and a technique for simulating software designs based on fuzzy logic. The result is the practical application of formal methods to a real-world software design. That is, we apply to an actual design for working software a formal method for validating the design and automating the analysis; and we do so in less time than it took to create the software design.

We start by specifying the design for a real-world problem of interest. We continue by using POAD Theory to structure an adequacy argument for the design. We apply a simulation technique based on fuzzy logic to fully justify our adequacy argument. Finally, we close with an analysis of our technique and conclusions about the significance of this research.

## II. STATE OF THE ART

Previous works like [5], [6] and [7] propose methods of pattern-based reasoning using rules generalized from specific design experiences. The works of [5] and [6] build rules from observed correlations between patterns and a particular software quality. The work of [7] places all possible pattern implementations into a limited set of categories, and then derives rules for each category. Works like [5], [6] and [7] required time-consuming tailoring of general rules before predictions could be made about real-world software designs. For example, the method described in [7] required users to make a complete formal model of a working system before it could be used to make predictions about the system's design. By contrast, we make predictions about software designs based on subjective arguments and use general rules only to structure those arguments and demonstrate validity. Our method allows us to make predictions about the consequences of a design with an effort much smaller than that required by previous works.

## III. A COLLABORATIVE SYSTEM DESIGN

Figure 1 is an example of a collaborative system [8] designed to report the environmental condition of a given region. Each sensor is capable of recording and reporting its local conditions, but to record and report the condition of the entire region requires that all sensor stations cooperate.

The nodes in the network are autonomous and spatially distributed across the region shown. Each sensor is capable of autonomously recording and reporting its local environmental conditions to a controller in its region. Task execution is distributed across multiple nodes since reporting conditions for the entire region requires that a controller communicate with multiple sensor stations. 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.

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. 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.

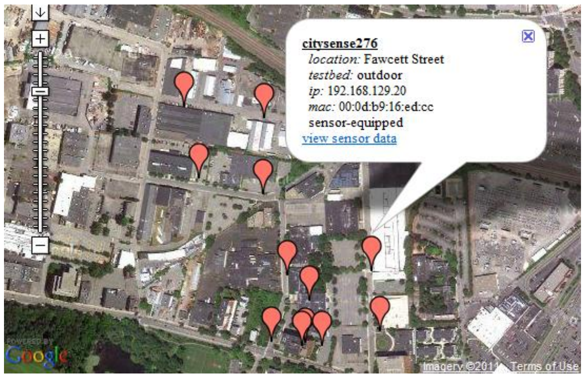

Figure 1: Example Collaborative System [9]

A robust design will allow the sensor stations (known as nodes) to both detect and mitigate these kinds of failures. 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 our design for a robust collaborative system. We consider Figure 2 a real-world design since it was taken from the design of an actual software system built to provide fault tolerance in collaborative systems [10]. Each *GroupNode* 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*. Using the JGroup communication API [11] the *PushPullStrategy* gives each *PushPullNode* the ability to communicate with other *PushPullNodes*.

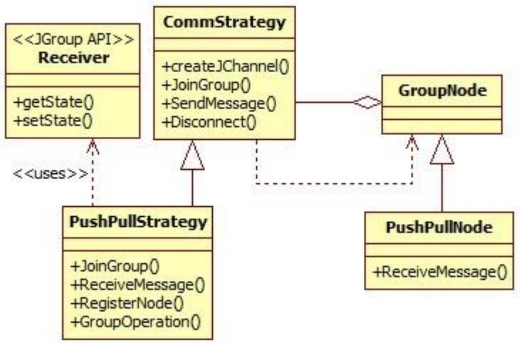

Figure 2: Design for robust collaborative system.

Figure 3 shows how the design works. The Controller relies on either sensor A or sensor B to report temperature for a given region. When the Controller wants a temperature reading from the zone, it joins the zone's group and executes the *commstrategy.GroupOperation()* operation. 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.

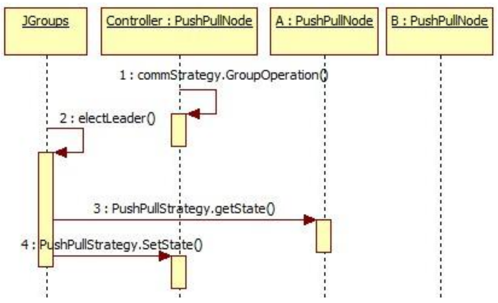
JGroups
Controller : PushPullNode
A : PushPullNode
B : PushPullNode
1 : commStrategy.GroupOperation()
2 : electLeader()
3 : PushPullStrategy.getState()
4 : PushPullStrategy.SetState()

Figure 3: Nodes participating in a group operation

We have a design, but is it adequate: does it solve our problem and satisfy our requirements? In the next section, we use the mathematics of POAD Theory to structure an adequacy argument for the design.

## IV. APPLYING PRACTICAL MATHEMATICS

POAD Theory is based on Problem-Oriented Software Engineering (POSE) [12] where engineering design is represented as a series of transformations from complex engineering problems to simpler ones. 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. The problem, $CSystem$, of designing a collaborative system can be expressed in POSE as:

$$CSystem{:}\, W,S \ \vdash R$$

(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.

POAD Theory uses POSE to represent software design patterns as justification for transforming a complex, unfamiliar problem into simpler, more familiar one [3], [4]. For example, the engineering expertise documented in the Object Group pattern describes how to achieve reliable multicast communication among objects in a network [14]. We can use the engineering judgment 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 $Obj$. We write this as

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

(2)

Equation (2) implies that if we have a solution to $Comm$ and $Obj$ then we also have a solution to $CSystem$. The reliable multicast communication that we get from the Object Group pattern may be sufficient to satisfy Req. 1 and Req. 2, but we have not yet addressed Req. 3. To satisfy Req. 3, we need to insulate $Obj$ so that it can continue to operate after losing communication with its environment. The Explicit Interface pattern describes how to achieve separation between an object and its environment [15]. We can use the Explicit Interface pattern $\ll EI \gg$ to justify the transformation of $Obj$ into a $Node$ that is separated from its environment by an interface $Intf$.

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

(3)

Equation (3) is a solution tree with $CSystem$ at the root. Two problem transformations extend the tree upward into 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 in (3) we could rewrite $Comm$ to appear as $\overline{Comm}$.

The argument represented by (3) is not complete and fully-justified until all leaf problems have been solved. We complete the argument by adding transformations and justifications sufficient to solve $Comm$, $Intf$, and $Node$.

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

(4)

In (4) the problems $Recvr$, $PPStrat$, and $PPNode$ correspond to the *Receiver*, *PushPullStrategy* and *PushPullNode* from Figure 2. 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. Although we already have justifications $EI$ and $OG$; justifications $J_1$, $J_2$, and $J_3$ are assumed, yet unknown. We can consider $CSystem$ solved by finding $J_1$, $J_2$, and $J_3$. In the next section, we use design simulation results to complete our missing justifications.

## V. APPLYING PRACTICAL SIMULATION

In this section, we use Fuzzy Inference [16] to simulate the effects of the *Receiver*, *PushPullStrategy* and *PushPullNode* (from Figure 2). Fuzzy inference is based on a generalized modus ponens [16] where arguments take the form:

$$\begin{gathered} If\ A\ Then\ B \\ A' \\ Therefore\ B' \end{gathered}$$

(5)

For example, suppose we accepted the general rule that: *if the Object Group pattern were 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.

In works like [17], [18], and [19] fuzzy logic has been used to reverse engineer design patterns: use fuzzy inference to determine if an existing solution, known to satisfy certain requirements, matches a general design pattern. In this work, we apply that idea 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.

We begin our simulation by creating fuzzy rules [16] that represent the design constraints of (3) and (4):

**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 *PushPullNode* communicates statically then degraded-mode operation will not be enabled

**Rule 5.** If the *PushPullNode* communicates dynamically and the Explicit Interface pattern is implemented then degraded-mode operation will be enabled

**Rule 6.** If the *PushPullNode* 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 [16] that allow the inference engine to turn the numeric values of the variables into the more intuitive concepts used in the Rules 1-6. For example, we defined membership functions (Poor, Good, and Moderate) for the Group Communication output variable so that a value of 0.7 would be considered mostly moderate, slightly good, and not at all poor. As shown in Figure 4, input variables representing our implementation choices are fed into an inference engine, which has been loaded with Rules 1-6. The inference engine calculates values for fuzzy output variables, which represent the results of the simulation.

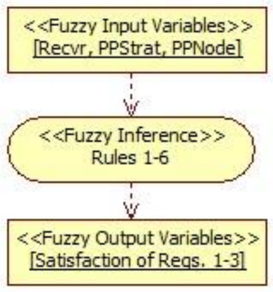
<<Fuzzy Input Variables>>
[Recvr, PPStrat, PPNode]
<<Fuzzy Inference>>
Rules 1-6
<<Fuzzy Output Variables>>
[Satisfaction of Reqs. 1-3]

Figure 4: Process flow of the simulation.

We simulate the design choices made in (4) by assigning specific values to the fuzzy input variables $Recvr$, $PPStrat$, and $PPNode$. Our variables range between 0 and 1 and we chose numeric values we believed best reflected our engineering judgment.

We assigned a value of 0.949 to the $Recvr$ variable because JGroups provides a faithful implementation of the Object Group pattern; we assigned a value of 0.762 to the $PPNode$ variable since we consider it a good approximation Object Group pattern's node element; and we assign a value 0.584 to the $PPStrat$ variable because we believe that it is not a very good representation of the intent of the Explicit Interface pattern.

The results of the simulation predict that the design decisions described in (4) and shown in Figure 2 will result in a collaborative system that satisfies Req. 1-3 (see Figure 5). The simulation predicts that the system will have good group communication (0.833), good fault tolerance (0.815), and will operate well in degraded mode (0.807).

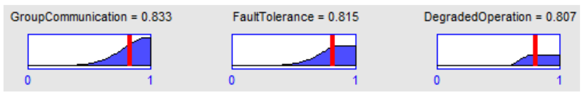

Figure 5: The results of collaborative system simulation.

The positive simulation results along with our analysis of the design spaces provides the justification ($J_1$, $J_2$, and $J_3$), needed to complete the argument (given by (3) and (4)) that the design shown in Figure 2 is adequate.

## VI. Analysis and Conclusions

Although the method we introduced does not provide us with proof that our design is adequate, it does provide us with a sound argument for a likely-stable design. Generalized modus ponens – the basis of fuzzy inference – is sound in that its conclusions are true if the premises are true [20]. We can trust the results of our simulation as long as we trust the rules that we establish for governing the simulation. Stable software designs tend to be built from stable sub-designs [21] – although we recognize that using stable sub-designs does not necessarily guarantee overall design stability. Because the mathematics that we use structures arguments based on software design patterns – known-stable designs – we have reason to believe that we are likely arguing for the adequacy of a stable design.

Our method is practical in that, with a relatively small amount of effort, we were able to use math and simulation to discover things about the design that are not obvious. With Eq. 1-5 and the associated explanatory text, we were able to create a mathematical model that had a meaningful correspondence to the collaborative system design in Figure 2. 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 (3) and (4); and the engineering expertise contained in the Object Group and Explicit Interface patterns; all we needed to validate the design of Figure 2 was to find justifications $J_1$, $J_2$, and $J_3$. We were able to provide that justification using Rule 1-6; fuzzy variable membership function definitions; and fuzzy inference.

Our use of math and simulation is, essentially, an application of analogical reasoning [22] where we draw a comparison between the design of Figure 2 and software design patterns. We were able to argue for the adequacy of our design by replacing the more difficult task of predicting the consequences of the design with the much easier task of comparing the design with known software design patterns. 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 simulation tells us just how close our design needs to be in order to produce satisfying results.

We started with a known software design, structured an argument for the adequacy of the design, and completed the argument using simulation. That order gave us a practical method of validation, but the individual methods are still valid even if we change the order. Suppose, instead, we started by structuring the argument, and then ran the simulation, and last found a software design that fit the argument and simulation. Instead of validating existing software, we would be predicting the existence of unknown software. We would have to simulate software that has not yet been designed; requiring us to guess at a likely implementation. In the future we will investigate if, by changing the order of our method (and overcoming the problems caused by that change), it is possible to use these same techniques to create an equally practical method of software design prediction.

## ACKNOWLEDGEMENTS

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] D. Jackson. *Lightweight Formal Methods*. FME 2001: Formal Methods for Increasing Software Productivity, Lecture Notes in Computer Science, Volume 2021, pp. 1, 2001

[2] 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, pp. 168-177, 1993.

[3] 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.

[4] 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, pp. 208-213, 2009.

[5] 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.

[6] 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), pp. 325–332, 2000.

[7] 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.

[8] 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, pp. 320-333, March 2004.

[9] http://www.citysense.net/, Last Accessed: May 2011

[10] J. Overton. *Collaborative Fault Tolerance using JGroups*. Object Computing Inc. Java News Brief, 2007, http://jnb.ociweb.com/jnb/jnbSep2007.html, Last Accessed April, 2011.

[11] The JGroups Project. http://www.jgroups.org/. Last Accessed April, 2011

[12] 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.

[13] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*, Volume 5. John Wiley & Sons, West Sussex, England, 2007.

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

[15] 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.

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

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

[18] C. De Roover, J. Brichau, and T. D'Hondt. *Combining fuzzy logic and behavioral similarity for non-strict program validation*. In Proc. of the 8th Symp. on Principles and Practice of Declarative Programming, pp. 15–26, 2006.

[19] I. Philippow, D. Streitferdt, M. Riebisch, and S. Naumann. *An approach for reverse engineering of design patterns*. Software Systems Modeling, pp. 55–70, 2005.

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

[21] R. Monson-Haefel. *97 Things Every Software Architect Should Know*. O'Reilly Media, Inc. 2009.

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