Communication Aspects with *CommJ*: Initial Experiment Show Promising Improvements in Reusability and Maintainability

Ali Raza
Computer Science Department
Utah State University
Logan, Utah, USA
ali.raza@aggiemail.usu.edu

Jorge Edison Lascano
Computer Science Department
Universidad de las Fuerzas Armadas
ESPE
Sangolqui, Ecuador
jelascano@dcc.espe.edu.ec

Stephen Clyde
Computer Science Department
Utah State University
Logan, Utah, USA
stephen.clyde@usu.edu

***Abstract*—A 2013 ICSEA paper introduced *CommJ* as an extension to *AspectJ* for encapsulating communication-related crosscutting concerns in modular, conversation-aware aspects. This paper now presents preliminary, but encouraging results from a subsequent study that shows six different ways in which *CommJ* can improve the reusability and maintainability of applications requiring network communications. We begin by defining a reuse and maintenance quality model as an extension to an existing quality model. We then identify six hypotheses that can be measured using metrics from the quality model. Finally, to test the hypotheses, we compare implementations of different sample applications across two study groups: one for *CommJ* and another for *AspectJ*. Results from the study show improvement in the *CommJ* for all six areas addressed by the hypotheses.**

Keywords-aspect-oriented programming (AOPL); crosscutting concerns; AspectJ; software reuse and maintenance; software metrics.

I. Introduction

Aspect-oriented Software Development (AOSD) first started to appear in the literature in 1997 [4][12] as a way of reducing the scattering and tangling of code caused by crosscutting concerns [15]. Its contribution was to encapsulate the essence of crosscutting concerns into abstractions, called *aspects*. An aspect is an Abstract Data Type (ADT) with all of the same capabilities as an object class, plus a few enhancements. Specifically, it can contain *advice*, which is logic for implementing crosscutting concerns that is automatically woven into appropriate places in the base applications. The *aspects* also include *pointcuts*, which describe where and when the advice weaving takes place. More specifically, each pointcut identifies a set of *joinpoints*, which are intervals in the execution of the system and weaving can occur before, after, or around these intervals [15].

AspectJ is an Aspect-oriented Programming Language (AOPL) that extends *Java* for aspects [14]-[17]. It allows programmers to *weave* advice into joinpoints that correspond to constructor calls or executions, methods calls or executions, class attribute references, and exceptions. The problem is that *AspectJ*, like other AOPLs, does not support the weaving of advice into high-level abstraction, like Inter-Process Communication (IPC) where each conversation has an independent context. IPC are ubiquitous in today's software systems, yet they are rarely treated as first-class programming concepts. Instead, developers typically have to implement communication protocols using primitive operations, such as *connect*, *send*, *receive*, and *close*. The sequencing and timing of these primitive operations can be relatively complex.

The *CommJ* framework (Section II) extends *AspectJ* so developers can weave crosscutting concerns into IPC in a modular and reusable way, while keeping the core functionality oblivious to those concerns. Specifically, it allows programmers to view individual conversations as uniquely identifiable concepts, with its own context and weave logic into a base application that makes use of the context information for individual conversations.

Our study investigates potential changes to the reuse and maintenance to software when developers use *CommJ.* It does so by evaluating certain desirable characteristics defining a quality model (Section III) that can be measured by computable metrics (Section IV). Based on initial theoretic notions, we hypothesize that developers should see reuse and maintenance improvements relative to six desired qualities (Section V) defined by the quality model. Section VI talks about our experiment methodology, which required formal approval from Institutional Review Board (IRB) [10], selection of the sample software application, and identifying interesting crosscutting concerns that would give us good coverage. The methodology also included typically, supporting activities such as recruitment and training of the developers. After the experiment, we collected data from the code, surveys, hourly journals, and questionnaires.

From the results (Section VII) of the study, we conclude that IPC software components developed with *CommJ* were more cohesive and oblivious. They were also less scattered, coupled, complex and smaller in size than similar components programmed in *AspectJ*. These preliminary results lead us to believe that further experimentation with *CommJ* and refinement of its framework could prove to be very beneficial to a wide range of software systems.

II. High-Level Overview of *CommJ*

CommJ enables the partitioning of a complex communication problem into manageable cohesive concepts and promotes greater reuse with better maintainability. Figure 1 shows an architectural block diagram that represents relevant conceptual layers and their dependencies. The following paragraphs describe these high-level components and their dependencies. More details on the architecture, design and examples are given in [1].

The lowest layer on the left is conceptual model, called the Universe Model for Communication (UMC). It is a formal description of common knowledge related to IPC. It describes, for example, the notation of a communication protocol in terms of role-specific state machines and message types. It then defines a conversation as an instance of two or more processes exchanging data according to the behavioral rules defined by a protocol. One important part of the UMC is the definition of message. Regardless of the system, every message is a uniquely identifiable thing (object) that is part of a conversation. How a system identifies messages and tracks their relationship to conversations are different, but the underlying concept is assumed to be true for systems that use IPC.

The next layer is the *Core CommJ Infrastructure.* It is an *AspectJ* library that defines message-event joinpoints and provides mechanisms to track conversations, which will hold value context information for communication aspects. A software developer that wants to use communication-related aspects simply has to include this library in the project.

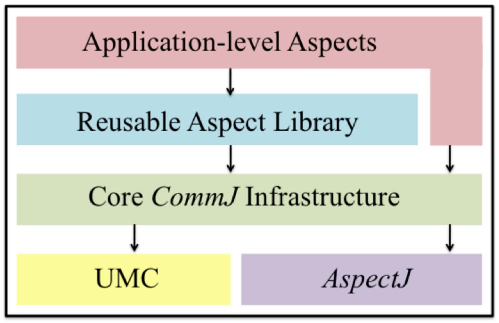

Figure 1. *CommJ* Architectural Block Diagram.

The *Reusable Aspect Library* (RAL) is a toolkit-like collection of communication aspects that application programmers should find useful for many different kinds of applications. They include aspects for measuring turn-around times, tracing conversations, and introducing behaviors into complex, multi-step protocols [1].

Application-level Aspects are those written by the application programmers, either using the abstractions provided by *CommJ* directly or by specializing the aspects in RAL. These aspects can encapsulate complex crosscutting behaviors in understandable and maintainable software components, without sacrificing obliviousness or flexibility.

III. EXTENDED QUALITY MODEL (EQM)

McCall identifies a list of eleven quality attributes [2], which have influence on quality of the software in general. Of these, we selected maintainability and reusability as the important qualities to consider initially because of potential for cost savings they both represent. Further work could focus on some of the other nine qualities.

To formalize the reuse and maintainability qualities, we adapt and extend the Sant'Anna quality model [3], because it allows for more generalized measurement, compared to Lopes' work [4] and it supports different types of implementation environments. The author builds the Quality model [3] using Basili's GQM Methodology [6]. Basili provides a three-step framework: (1) list the major goals of the empirical study, (2) derive from each goal the questions that must be answered to determine if the goals have been met; (3) decide what must be measured in order to be able to answer the questions adequately. In a nutshell, the model consists of *Qualities*, *Factors*, *Internal Attributes*, and *Metrics* (see Figures 2 and 3 for more details.).

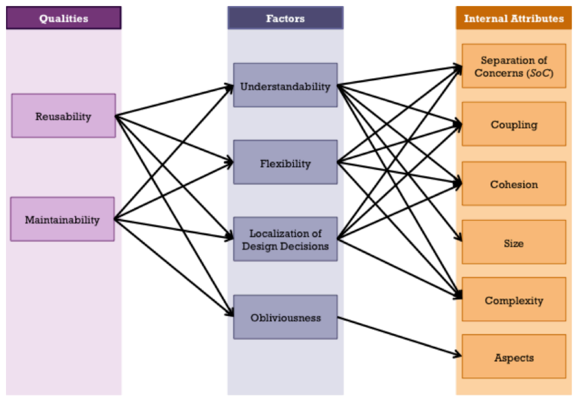

Figure 2. Extended Quality Model (*EQM).*

The qualities, such as reusability and maintainability, are the most abstract of the concepts in the model and represent the ultimate goals of "good" software. Each quality is determined by one or more factors, which are in turn determined by internal attributes. Although still abstract, these internal attributes are properties related to well-established software-engineering principles and there exists some informal notations on how to assess or evaluate them. And, that's where the metrics come in. The metrics means of measuring the internal attributes, or at least giving them a rough relative ranking. Ideally, we would like to be able to compute all metrics automatically, but that is not mandatory.

In our EQM [3], localization of design decisions, and code obliviousness were not part of original quality model [3]. However, we introduced them in our EQM for two reasons. Firstly, Parnas [27], in his landmark paper proposes three important characteristics of modular code, which were understandability, flexibility, and localization of design decisions (information hiding). Hence, reasoning maintainability and reusability only in terms of understandability and flexibility is not complete. Introduction of obliviousness is also equally important. By the time Parnas proposed the definition of modular code, obliviousness had not been invented as a fundamental design principle. However, in the context of our research experiment, which depends heavily on measuring crosscutting concerns, code obliviousness becomes very critical.

IV. EQM METRICS

The EQM includes 16 metrics for the six different internal attributes shown in Figure 3. Ten of the metrics can be computed automatically [20] from the code written by the subjects. The others have to be computed by hand. Below are brief descriptions of these metrics, so the reader can better understand the results presented in Section VII.

A. *SoC Metrics*

Separation of Concerns (SoC) defines ability to identify, encapsulate and manipulate those parts of software that are relevant to a particular concern [23]. Concern Diffusion over Application (CDA) and Concern Diffusion over Application Operations (CDO) are the two SoC metrics. CDA counts the number of primary components (class or aspect) whose main purpose is to contribute to the implementation of a concern. CDO counts the number of primary operations and advices that contribute to the implementation of a concern.

B. *Coupling Metrics*

Coupling is an indication of the strength of interconnections between the components in a system [24]. The EQM describes three coupling metrics. First, Coupling between Components (CBC) counts the number of other classes and aspects to which a class or an aspect is coupled. Excessive coupling of concerns increases CBC, which can be detrimental to the modular design and prevent reuse & maintenance. Depth Inheritance Tree (DIT) counts how far down in the inheritance hierarchy a class or aspect is declared. As DIT grows, the lower-level components inherit or override many methods and leads to design complexity and understanding problems. Number of Children (NOC) counts the number of children for each class or aspect. As NOC increases, the abstraction represented by the parent component can be diluted.

C. *Cohesion Metrics*

The cohesion of a component is a measure of the closeness of relationship between its internal components [24]. Lack of Cohesion in Operations (LCO) is the only cohesive metric in EQM that measures the cohesion of a class or aspect in our model. It does so in terms of number of method and advice pairs that do not access the same instance variable and hence should be separated.

D. *Size Metrics*

Size metrics physically measure the length of a software system's design and code [25]. EQM describes the following six size related metrics. Lines of Code (LOC). The greater the LOC, the more difficult it is to understand and manage the software. Method lines of Code (MLOC) is the average number of the lines of code per method. Kemerer [9] states that the greater the MLOC for a component, the more complex the component would be. Number of Operations (NO) counts the number of operations in a component. Objects with large number of operations are less likely to be reused. Number of Parameters (NP) counts the number of parameters for methods in each class or aspect. A method with more parameters is assumed to have more complex

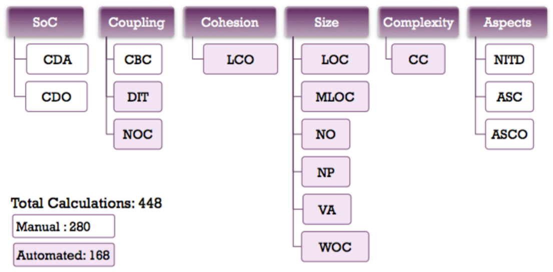

Figure 3. Measurement Metrics in *EQM.*

collaborations and may call many other method(s). Vocabulary Size (VA) counts the number of system components, i.e., the number of classes and aspects into the system. Sant'Anna [3] claims that if VA increases, it is an indication of more cohesion and less tangling for set of ADTs. Finally, Weighted Operations per Component (WOC) metric measures the complexity of a component in terms of its operations. The operation size measure is obtained by counting the number of parameters of the operation. An operation with more parameters than another is likely to be less understandable.

E. *Complexity Metric*

Complexity measures how components are structurally interrelated to one another. EQM uses Cyclomatic Complexity (CC) for measuring the complexity of the program. Mathematically, the cyclomatic complexity of a structured program is defined with reference to the control flow graph of the program. The metric is defined by the number of independent paths and provides an upper bound for the number of test cases that must be conducted to ensure that all statements have been executed at least once. A high value of CC affects program maintenance and reuse.

F. *Obliviousness (Aspects) Metrics*

Obliviousness is the idea that core functionality should not have to know about crosscutting concerns [13]. EQM defines three quality metrics for obliviousness. First, Number of Inter-type Declarations (NITD). A higher value of NITD indicates a tighter coupling between the aspect and application components. Second, Aspect Scattering over Components (ASC) counts the number of aspect components scattered over application components. It measures the tangling of aspects in the application components. More tangling of aspects in the program makes the original application less reusable and maintainable. Finally, Aspect Scattering over Component Operations (ASCO) counts the number of aspect components scattered over application component operations. ASC gives a high-level overview of the application tangling in the aspect components but ASCO provides more insight on operations-level tangling of applications inside aspect components.

V. HYPOTHESIS

The theoretical ideas that underpin *CommJ* lead to the following six hypotheses, with respect to comparing the reusability and maintainability of IPC software built with *CommJ* instead of just *AspectJ*.

- *Hypothesis 1:* If crosscutting IPC concerns are effectively encapsulated in *CommJ* aspects, then the software has better *separation of concerns* and less scattering (as described by CDA, CDO in Section IV.A than equivalent systems developed with AOP design techniques.
- *Hypothesis 2:* If crosscutting IPC concerns are encapsulated in *CommJ* aspects, then the software has *lower coupling* (as described by CBC, DIT, NOC in Section IV.B) than equivalent systems developed with AOP design techniques.
- *Hypothesis 3:* If crosscutting IPC concerns are encapsulated in *CommJ* aspects, then the software has *higher cohesion* and less tangling (as described by LCO in Section IV.C) than equivalent systems developed with AOP design techniques.
- *Hypothesis 4:* If crosscutting IPC concerns are encapsulated in *CommJ* aspects, then the software is not significantly *complex* (as described by CC in Section IV.D) than equivalent systems developed with AOP design techniques.
- *Hypothesis 5:* If crosscutting IPC concerns are encapsulated in *CommJ* aspects, then the software is significantly more *oblivious* (as described by NITD, ASC, ASCO in Section IV.E) than equivalent systems developed with AOP design techniques.
- *Hypothesis 6:* If crosscutting IPC concerns are encapsulated in *CommJ* aspects, then the software is not significantly larger (as described by LOC, MLOC, NO, NP, VA, WOC in Section IV.F) than equivalent systems developed with AOP design techniques.

VI. Experiment Methodology

The research experiment consisted of the following steps:

A. Experimental Approval

In the first step, we submitted an application for conducting this Human Research Experiment to the IRB [10] and got its approval. All the researchers then passed the online human research experiment-training course offered through Collaborative Institutional Training Initiative (CITI) [11].

B. Selection of Applications and Crosscutting Concerns

We selected applications that were multithreaded, used whether JDK sockets or channels. The applications were diverse in the way they implemented IPC and therefore provide good coverage of different types of communication heterogeneities. Finally, each application supported more than one communication protocol. Table 1 lists the set of selected applications.

Since the experiment would eventually require developers to modify or extend applications for requirements that represented communication-related crosscutting concerns, our methodology included a step, which systematically selected our representative crosscutting concerns. Developers would have to apply each of these to the applications, individually. Additionally, to minimize noise in our data, we wanted to make sure that these crosscutting concerns were sufficiently simple that a novice programmer could understand the need and come up with a solution in less than 10 hours. Table 2 introduces the set of selected crosscutting concerns.

C. Recruitment and Training of Participants

To transparently recruit the candidates, we sent invitation letters and recruited seven volunteer developers who were experienced in object-oriented software development, Java and software-engineering design principles such as modularity and reusability. We then randomly organized them into two study groups: A and B. Group A programmed using an AOP approach and Group B used *CommJ*. Next, the participants completed a survey that assessed their background and skill levels. We also provided AOP training to developers in Group A, and had them worked through some practice applications. Similarly, we trained Group B

TABLE I. SELECTED SAMPLE APPLICATIONS

Application Name	Description
Levenshtein Edit-Distance Calculator (LD)	A server will calculate the LD between two input strings, provided by the client, over a connection-oriented communication.
File Transfer Program (FTP)	A file transfer protocol over connection-oriented communication.
Weather Station Simulator (WS)	A simple weather station simulator, supported by a Transmitter and a Receiver.

TABLE II. SELECTED CROSSCUTTING CONCERNS

Application Name	Description
Version Compatibility	This concern adapted one version of the message to another, so processes running different versions could still communicate with each other. The crosscutting concern included knowledge of converting one version to another and conversely
Symmetric-Key Encryption	It encrypted the communication between a sender and receiver using symmetric-key encryption
Measuring Performance	It measured some performance related statistics for message-based communications between sender and receiver

developers with *CommJ*, and had them worked through some practice applications.

D. *Experiment Phases*

In the first phase, participants filled a pre-implementation questionnaire, developed the application using initial requirements, recorded hourly journals and completed a post implementation questionnaire. In the second phase, we requested enhancements (sample applications and crosscutting concerns), had them revised their implementation accordingly, and then collected those software systems. Participants again completed the pre and post questionnaire and wrote their experiences in the hourly journals.

Finally, after the second phase, we analyzed and evaluated the reusability and maintainability using various software artifacts, which included surveys, questionnaires, hourly journals, and actual code.

We used both manual computation and automated tools to compute measurements for all 16 metrics [20]. Experiment generated a total of 28 software systems. With 16 code metrics in the EQM, we had a total of 448 measurements, 280 computed automatically with a tool [20] and 168 calculated manually.

VII. Results

This section presents the data collected from the experiment and our results in context of the six hypotheses. In the following graphs, the vertical axes represent the measurements, and the horizontal axes represent the four activities of the experiment. For each activity there are two bars: a blue bar is for the results of *AspectJ* group and a green bar for *CommJ* group.

A. *Hypothesis 1: Better Separation of Concerns*

From the graph in Figure 4, we found that CDA and CDO values for the *CommJ* group went to zero in all four activities of the experiment. The reason for this phenomenon is that *CommJ* pointcuts provide total obliviousness between the application and communication-related crosscutting concern. *AspectJ*, components and their operations for crosscutting concern were significantly more diffused in the application because the pointcuts had to be tied to programming constructs instead of communication abstractions. From these results, we can conclude that Hypothesis 1 holds true for better separation of concerns in *CommJ* than in *AspectJ*.

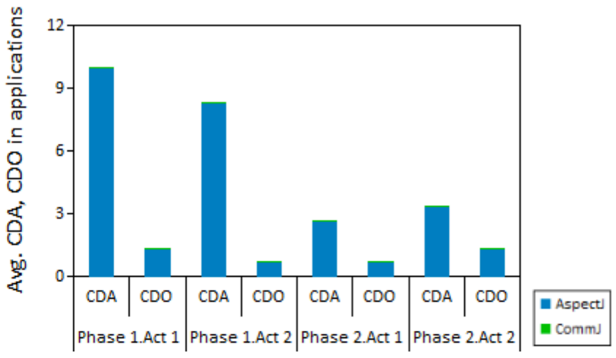

Figure 4. CDA, CDO coverage over phases.

B. *Hypothesis 2: Reduced Coupling*

The graph in Figure 5 indicates that *CommJ* implementations significantly reduced the values of CBC, DIT and NOC as compared to *AspectJ* implementations. *CommJ* crosscutting concerns didn't maintain any direct relationship with the application components and thus had a lower CBC value. However, in *AspectJ*, excessive coupling of concern with the application increased CBC, which hindered reuse and maintenance.

The reason for higher DIT and NOC values in *AspectJ* was that the participants preferred to override parent methods in crosscutting concerns to share data structures across aspect and application components during message passing. However, *CommJ* provides comprehensive set of pointcuts that fully encapsulates the IPC abstractions and thus participants didn't need to override or inherit the aspects.

From these results, we can conclude that Hypothesis 2 holds true for reduced coupling in *CommJ* than in *AspectJ*.

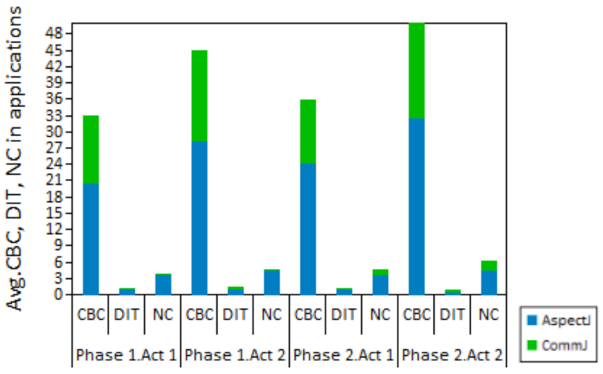

Figure 5. CBC, DIT, NC coverage over phases.

C. *Hypothesis 3: Improved Cohesion*

The results from the graph in Figure 6 demonstrate that *CommJ* maintains a lower value for LCOO than *AspectJ* in all phases of the experiment. Sant'Anna [3] says that LCO measures the degree to which a component implements a single logical function. Results argue that *CommJ* implementations are more cohesive and logical than *AspectJ*,

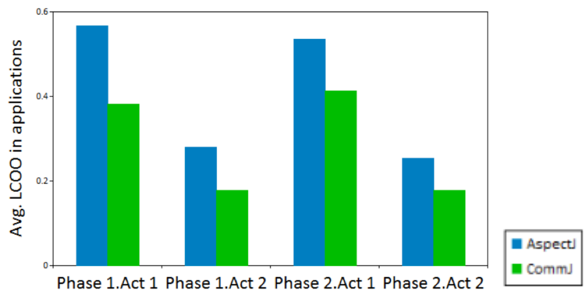

Figure 6. LCOO coverage over phases.

hence have a lower LCO value, which concludes that Hypothesis 3 holds true for increased cohesion in *CommJ* than in *AspectJ*.

D. Hypothesis 4: Reduced Complexity

The graph in Figure 7 shows that value of CC is smaller for *CommJ* than *AspectJ*, because *CommJ* hides complex IPC abstractions, which results in simple conditional statements and less tangled code. From these results, we can conclude that Hypothesis 4 holds true for less complex software in *CommJ* than *AspectJ*.

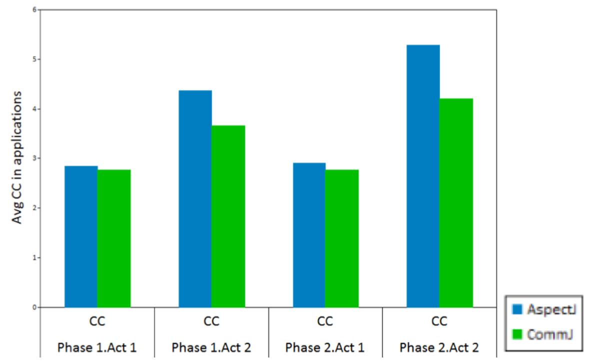

Figure 7. CC coverage over phases.

E. Hypothesis#5: Improved Obliviousness

The following graph in Figures 8 shows that *CommJ* implementations significantly reduced the values of NITD, ASC and ASCO metrics.

The reason for having a zero value for NITD in *CommJ* was that the participants used IPC constructs and did not need to use inter-type declarations (ITD) for sharing of data structures between application and aspect component. Significant reduction in ASC and ASCO was due to the layers of indirection between the application and aspect components, which *CommJ* provides but missing in *AspectJ*.

From these results, we believe that Hypothesis 5 holds true for less oblivious software concerns in *CommJ* than *AspectJ*.

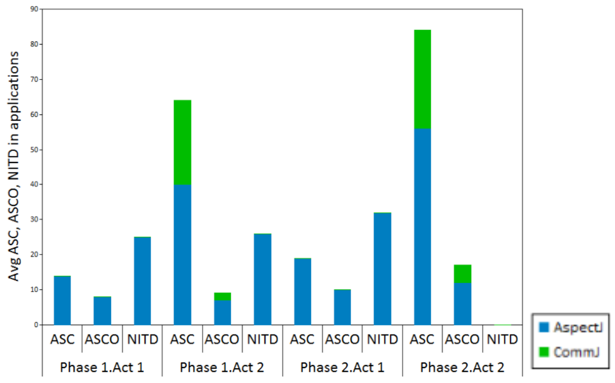

Figure 8. ASC, ASCO, NITD coverage over phases.

F. Hypothesis#6: Reduced Size

The graphs in Figure 9 shows that *CommJ* implementations significantly reduced the metrics values for LoC, MLoC, NP, NO and WOC and increase for VA in all phases of the experiment.

In comparison with *AspectJ*, *CommJ* participants found better pointcuts that helped them code the crosscutting concerns with less LOC. This is because the UMC models various general network and distributed abstractions. *CommJ* captures those abstractions in meaningful, reusable joinpoints and a family of base aspects, which helped the participants implement the application crosscutting concerns in simpler units, with no extra lines of code and fewer operations. Hence, *CommJ* reduced MLOC, NO, NP and WOC. Finally, the VA results indicate that average VA for all programs was more for *CommJ* than *AspectJ*, which, as Sant'Anna [3] claims, is an indication of more cohesion and less tangling. From these results, we can conclude that Hypothesis 6 holds true.

Besides analysis of the hypotheses via the metrics, we also collected observations through participant questionnaires and daily journals. On writing clean code, we found that 100% of *AspectJ* participants in the Phase 1 were struggled with identifying meaningful pointcuts for implementing the add-on requirements, while 33% of them still struggled with the same issue during Phase 2. On the other side, none of the *CommJ* participants struggled with this problem in either phase, which seem to indicate that *CommJ* provides simple pointcuts for IPC abstractions.

On reusability, we observed that 67% of the *AspectJ* participants in Phase 1 agreed that their applications might not run after removing the extension part from the original application. This percentage further increased to 100% in Phase 2. On the other hand, none of the *CommJ* participants felt this way for either phase. Similarly on maintainability, 100% of the *AspectJ* participants said that their changes (for either phase) introduced new dependencies in the original sample application. However, none of the *CommJ* participants felt the same way. The survey also provided some anecdotal information on frequency of bugs, specifically 67% of the participants in *AspectJ* group said that their implementation of extensions introduced new bugs in Phase 1. This percentage further increased to 100% in Phase 2. However, only 25% of the *CommJ* participants felt that their extensions introduced bugs in Phase 1 and Phase 2. This tells us that *CommJ* modularization and obliviousness may decrease the introduction of failures and the debugging time.

G. Threats to the Validity

Despite our best effort to perform the experiment objectively with minimize extraneous variables, it is important to recognize that this preliminary study has some significant threats to validity. These include variations in intelligence among the developers, health factor, work environment, and personnel commitment. Still, we believe that the results are very encouraging.

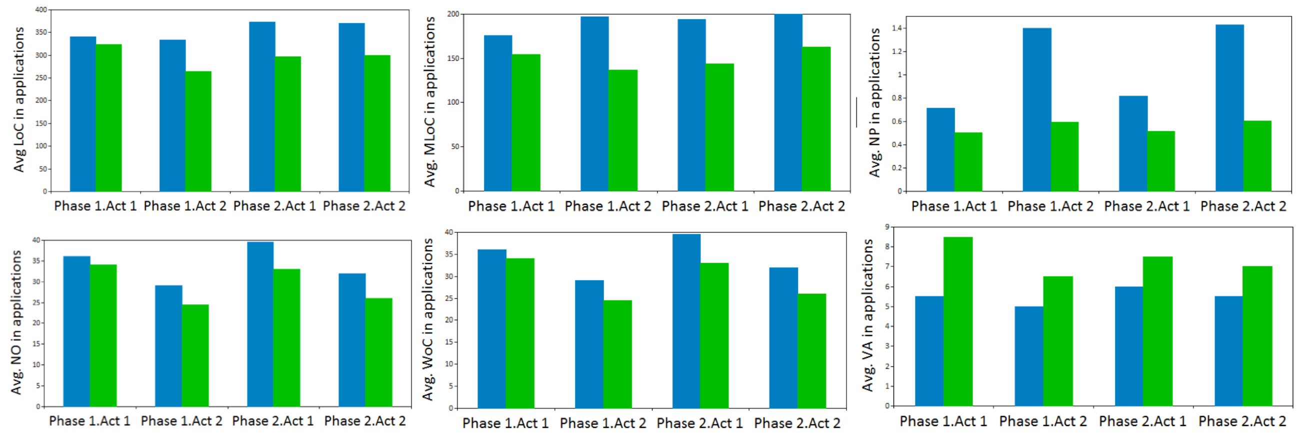

Figure 9. LoC, MLoC, NP, NO, WoC coverage over phases.

VIII. SUMMARY AND FUTURE WORK

In ICSEA 2013, we presented the design and implementation of a new AOPL framework, called *CommJ*, which allows developers to encapsulate IPC crosscutting concerns in reusable and maintainable modules [1]. This paper discusses an initial study on hoped-for benefits of *CommJ* in comparison with *AspectJ*. It defines an extended quality model, then setup an experiment methodology, involving six quality hypotheses and data collection from 28 programs. The results from this preliminary investigation provides sufficient evidence to conclude that *CommJ* is capable of encapsulating a wide range of communication-related crosscutting concerns and that it can provide better maintainability and reusability. In the future, we plan to conduct additional studies, refine the *CommJ* Infrastructure, and extend the library of reusable aspects (RAL).

REFERENCES

[1] A. Raza and S. Clyde, "Weaving Crosscutting Concerns into Inter-Process Communication (IPC) in *AspectJ,"* in ICSEA 2013, Venice, Italy, pp. 234-240.

[2] Jim A. McCall, "Factors in Software Quality," in Nat'l Tech. Information Service, 1977, vol. 1, 2 and 3.

[3] C. Sant'Anna, A. Garcia, C. Chavez, C. Lucena, and A. Von Staa, "On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework," in 17th Brazilian Symposium on Software Engineering (SEES 2003), Manaus, Brazil (2003), PUC-RioInf.MCC26/03.

[4] C. Lopes, "D: A Language Framework for Distributed Programming," in PhD Thesis, College of Computer Science, Northeastern University, 1997.

[5] J. Zhao, "Towards a Metrics Suite for Aspect-Oriented Software," in Technical-Report SE-136-25, Information Processing Society of Japan (IPSJ), March 2002.

[6] V. Basili, G. Caldiera, and H. Rombach, "The Goal Question Metric Approach," in Encyclopedia of Soft. Eng., September 1994, vol. 2, pp. 528-532, John Wiley & Sons, Inc.

[7] L. Benavides, M. Sudholt, W. Vanderperren, B. Fraine, and D. Suvee, "Explicitly distributed AOP using AWED," in AOSD 2006, pp. 51-62.

[8] G. Kiczales and M. Mezini, "Aspect-Oriented Programming and Modular Reasoning," in ICSE 2005, pp. 49-58.

[9] S. Chidamber and C. F. Kemerer, "A Metrics Suite for Object-Oriented Design," in IEEE Trans. Software Engineering, June 1994, vol. SE-20, No. 6, pp. 476–493.

[10] Institutional Review Board (IRB), http://rgs.usu.edu/irb, retrieved: August, 2014.

[11] Collaborative Institutional Trainig (CIIT), https://www.citiprogram.org, retrieved: August, 2014.

[12] G. Kiczales et al., "Aspect-oriented programming," in (ECOOP), 1997, pp. 220—242.

[13] L. Bergmans and M. Aksit, "Composing Software from Multiple Concerns: Composability and Composition Anomalies," in ICSE'2000. Position paper.

[14] AspectWorkz2, http://aspectwerkz.codehaus.org, retrieved: August, 2014.

[15] ApectJ, http://www.eclipse.org/*AspectJ*, retrieved: August, 2014.

[16] JBoss AOP, http://www.jboss.org/jbossaop, retrieved: August, 2014.

[17] Spring AOP,org.springframework, retrieved: August, 2014.

[18] C. Clifton and T. Leavens, "Obliviousness, Modular Reasoning, and the Behavior Subtyping Analogy," in SPLAT 2003.

[19] R.D. Tennent, "The Denotational Semantics of Programming Languages," in Communications of ACM 1976, pp. 437-453.

[20] Metrics plugin, http://metrics2.sourceforge.net, retrieved: August, 2014.

[21] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, "Distributed Systems: Concepts and Design (5th ed.)," in 2011, Addison-Wesley Publishing Company, USA.

[22] R. Dromey, "A Model for Software Product Quality," in IEEE Transactions on Software Engineering," in February 1995, vol. 21, No. 2, pp. 146-162.

[23] P. Tarr and S. Sutton, "N-Degrees of Separation: Multi-Dimensional Separation of Concerns," in 21st International Conference on Software Engineering, May 1999, pp. 107-119.

[24] I. Sommerville, "Software Engineering", 6th Edition, Harlow, England. Addison-Wesley. 2001.

[25] N. Fenton and S. Pfleeger, "Software Metrics: ARigorous and Practical Approach," in 2.ed. London: PWS. 1997.

[26] A. Raza,. "Improving reuse and maintenance of communication softwares with conversation-aware aspects," in Ph.D. Disseration, Computer Science Department, Utah State Univeristy 2014.

[27] D. Parnas, "On the criteria to be used in decomposing systems into modules," in Communications of the ACM 15, val. 12 (December 1972), pp. 1053-1058.