Measuring the Complexity of Mobile Agents Designed with Aspect/J

Jana Dospisil and Arin Khemngoen
Monash University, Australia

jana.Dospisil@infotech.monash.edu.au akhe2@student.monash.edu.au

Abstract

This paper describes research in measuring the code complexity of mobile agent applications designed with aspect-oriented programming (AOP) as captured in the AspectJ™ language. The modularized code encapsulating agent interactions is characterized by class hierarchies which are shared structures. Mobile agent design suffers from frequent changes in interaction protocols which leads to chaotic development. Additional subclassing, modification to protocols, restructuring of the class hierarchies, changes to visibility of attributes and methods overloading result in increased complexity of the code and disorder. Our experonce with fine tuning of protocols shows that the probability that a subclass will not consistently extend the protocol content of its superclass is increasing with the depth of hierarchy. The tools like *Hyper/J* and *Aspect/J* support the separation of concerns thus allowing different approach to evolving the protocol content rather than extending the class hierarchies. In this paper we present the approach to analyzing protocol design and assessing the complexity by measuring the entropy of the mobile agent application code designed with Aspect/J. The comparison of complexity measures with the same mobile agent application designed and maintained as typical Java application indicates reduction in complexity in favor of design with Aspect/J.

Keywords: mobile agent, contract net protocol, complexity, entropy based metrics, separation of concerns

Introduction

Agents are viewed as the next significant software abstraction, and it is expected they will become as ubiquitous as graphical user interfaces are today. It is envisaged that the decision making processes and interactions between agents will be very fast (Kephart et.al., 1998) and the agents will be characterized by fast growing complexity of the code. The new approach to development is separation of concerns and relevant tools such as Aspect/J (Kiczales et.al., 1997) or Hyper/J (Ossher & Tarr, 1998) which facilitate the development process (Kendall, 1999, Dospisil, 2003).

The source of the problem in software development is that some kinds of behaviour or functionality *cross cut* or are *orthogonal to* classes in many object-oriented components and they are not easily modularised to a separate class. Examples of such behaviour include the following: synchronization and concurrency, performance optimization, exception handling and event monitoring, coordination and interaction protocols, and object views. To measure the quality of separation either in *N-dimensional* space or even the orthogonal separation only as seen in *Aspect/J*, the new set of complexity metrics is required.

Paper Accepted as a Regular Paper

Layout of the paper: Section 2 provides a brief overview of negotiation protocols and complexity measures. Section 3 discusses the use of AOP in mobile agent design. Section 4 provides a brief overview of entropy metrics used to measure complexity of design. Section 5 contains some results of our experiment.

Background

Negotiation Protocols

Negotiation is a search process. The participants jointly search a multi-dimensional space (e.g. quantity, price, and delivery) in an attempt to find a single point in the space at which they reach mutual agreement and meet their objectives. For many-to-many coupling or interaction between participants, the market mechanism is used, and for one-to-many negotiation, auctions are more appropriate. The market mechanism often suffers from an inability to scale down efficiently (Osborne & Rubinstein, 1990) to smaller numbers of participants. One-to-many interactions are influenced by strategic considerations and involve integrative bargaining where agents search for *Pareto efficient* agreement.

Many different types of contract protocols (cluster, swaps, and multiagent, as examples) and negotiation strategies are used and have been experimentally implemented (Sandholm & Lesser, 1998; Mass-Colell, Whinston & Green, 1995 and others). Agents based on constraint technology use complex search algorithms to solve optimization problems arising from the agents' interaction. In particular, coordination and negotiation strategies in the presence of incomplete knowledge are good candidates for constraint-based implementations (Nareyek, 1998)

Contracts in automated negotiations consisting of self-interested agents are typically designed as binding (impossible to breach). In cooperative distributed problem solving, commitments are often allowed to be broken based on some local reasoning. Frequently, the protocols use continuous levels of commitment based on a monetary penalty method (Sandholm & Lesser, 1998). The inflexible nature of these protocols restricts an agent's actions when the situation becomes unfavourable. The models, which incorporate the possibility of decommitting from a contract with or without reprisals (Sen, & Durfee, 1994 and Smith, 1980) can accommodate some changes in the environment and improve an agent's gain. The design and coding of these protocols is typically too rigid with respect to evolving, and accommodation dynamic requirements for improvement, in particular in mobile environment. The complexity of these protocols and large number of exceptions which must be handled is the main reason why the resulting code shows entropic tendencies by deepening of hierarchies and extensive subclasses.

Aspect-based static agent design has been described by Kendall (1999). She proposed role based design with aspects representing roles. In this approach, a role is a module equipped with rich interface that can be plugged in and out of an application, as needed. She used Line of Code (LOC) measure to assess the final code complexity. Our experiments with negotiating mobile agents indicate that such metrics are insufficient, and lack interpretation framework.

Overview of Entropy Based Complexity Measures

Entropy-based complexity measures are based on the theory of information. This is the approach taken by Davis and LeBlanc (1988) who quantify the differences between *anded* and *neted* structures to provide an unbiased estimate of the probability of occurrence of event *m*. This measurement is based on chunks of FORTRAN and COBOL code (represented by nodes in the DAG) with the same in-degree and the same out-degree to assess syntactic complexity.

Belady and Lehman (1976) elaborated on the law of increasing entropy: the entropy of a system (the level of its unstructuredness) increases with time, unless specific work is executed to maintain or reduce

it. Entropy can result in severe complications when a project is modified, and it is generally an obstacle to maintenance.

The use of entropy as a measure of information content, introduced by Harrison, has been around since 1992 (1992). Harrison's software complexity metric is based on empirical program entropy. Harrison assessed the performance of these entropic metrics in two commercial applications written in C language with the total number of lines of code being over 130,000.

The work of Bansiya and Davis (Bansiya , Davis, & Etzkorn, 1999) introduces a similar complexity measure – *Class Definition Entropy* (*CDE*) – replacing the operators of Harrison with name strings used in a class. The assumption that all name strings represent approximately equal information is related to the possible error insertion by misusing the string. The metric has been validated on four large projects in C++ and results have been used to estimate a *Class Implementation Time Complexity* measure

Utilizing AOP in Mobile Agent Design

In aspect-based design of negotiating agents, we separate two entities: *negotiation protocol* and *negotiation strategy*. *Negotiation strategy* can be seen as a private formalized strategy responsible for computation of appropriate actions and outcomes. The generated actions and outcomes depend on the role the agent assumes, the negotiation protocol used, and agent's relationship with other parties (e.g. cooperating self-interested agent).

Negotiation protocols together with other constraints restrict interaction between participating agents. Each protocol follows the rules outlined below:

1. A goal which is associated with a set of parameters (attributes)
2. Relationships among participants
3. Timeout and termination conditions
4. Binding rules with regard to penalty, commitment and decommitment.

We need to map the above rules into a set of request and response messages and a set of interfaces which each agent has to implement. The format of messages (ontology) and the interfaces to support the protocols must be specified and disclosed. If the selected strategy is not producing the desired results, the agent should be able to move to a different strategy while still conforming to the given negotiation protocol. Alternatively, the negotiation can be terminated and restarted with a new protocol and strategy providing the involved participants reaches mutual agreement in this matter. The implementation of improved protocol performance is typically based on changes to class hierarchies and method overloading or overriding. To avoid deepening of class hierarchies and other changes, we propose to build negotiation protocols and strategies as replaceable modules encapsulated in aspects.

Brief Overview of Entropy Based Metrics for Aspect/J

With Aspects/J specific features, we focus on reduced complexity as the affect of splitting the control-flow to multiple streams. Control flow in a program is the order of in which contained sequential units of code is executed. The local code within a unit (method) does not alter interaction between objects; therefore, it does not contribute significantly to complexity flow. Control flow is then a measure of object interaction.

The entropy based metrics have been proposed and described in (Dospisil, 2003) and tested on small example of Java code. The primary lexical abstraction in Java is a symbol (called either name or identifier). It is represented by a character string with the following properties: scope (private, protected, public, and package), type and storage class (class variable or method, import). Strictly local symbols are

excluded from the model: local variables within methods and parameter names in method signature do not alter object control flow. Primary lexical abstraction in Aspect is the symbol (keyword) and name-space for pointcuts.

Data collection and extraction of symbols representing the control flow has been done with concern graphs and the FEAT tool (Murphy, Lai, Walker, & Robillard, 2001). The structural system model is composed of different types of nodes and edges:

- *Symbol nodes* s_i are end nodes that correspond to global symbols s in the class stream,
- *Class nodes* c_j are represented by the top structural units from which all derived nodes are sourced

Edges represent dependencies between the following elements:

- *class nodes* and *aspects* that describe the dependency between aspects and classes using the particular aspects. *Aspect dependency* refers to treatment of crosscuts.
- *symbols and nodes* which directly provide the source for symbol. *Symbol dependency* refers to dependencies relevant to processing logic.

Definitions and symbols used in equations are in Table 1:

The total entropy of the system is given by the equation:

$$H(P) = H(S) + \frac{V_s}{V_e} H(E)$$

$V_s = \sum_{i=0}^{I} s_i$	Total number of all **symbol nodes** in the graph. We assume that the number of symbol nodes is within the range 0 to I-1.
$V_c = \sum_{j=0}^{C} c_j$	Total number of class nodes range from 0 to C-1
$V_e = V_d + V_a$	Number of all **edges** in the graph (aspect edges - and symbol edges) relevant to the particular symbol and aspect.
$V_d = \sum_{i=0}^{I} e_i^s$	Total number of dependency edges in the class stream traversed to a symbol *s (for0<s<=S).* Assumption: the symbol s may occur in multiple nodes.
$V_a = \sum_{a=0}^{A} e_a^c$ $p(e_i) = \frac{V_d}{V_s}$	Total number of edges with aspects that have an entry in class *c* (e_a^c → a[th] aspect edge associate with class c); the aspect is used by multiple classes) Probability that *i*[th] symbol node is sourced by *e* edges
$p(e_a) = \frac{V_a}{V_d}$	Probability that a random symbol node *i* has an aspect associated with it

$p_e(d)$	Probability that the dependency edge has length d (the number of nodes needed to be traversed to reach the edge e is d)
$p(d)$	Probability that two random symbol nodes i will have d distance between them.
$H(S) = -\sum_{i=1}^{M} p(e_i)\log_2 p(e_i)$)	Entropy of the total number edges which serve as information source to a symbol node
$H(E) = \sum p(d)\log_2(V_s p(d))$	Entropy of finding the edge destination if we know the starting point (symbol source)
$H(A) = -\sum_{i=1}^{M} p(e_a)\log_2 p(e_a)$	Entropy of aspects

Table 1: Definitions and equations

Weighted Entropy

During our experiment with agent application we established that not all types of edge are equally important for given symbols. In order to distinguish the dependency edges according to their importance with respect to a given qualitative characteristics we have assigned to each edge type a non-negative weight proportional to its importance and significance.

Weights of every edge type have an objective character representing the ratio of the objective probability that the edge path is the source of information for symbol s.

$$w(e_i^n) = -\frac{q(e_i^n)}{\log_e q(e_i^n)}$$

In this case we obtain the following expression for weighted entropy:

$$H(e_i^n) = \sum_{i=1}^{I} q(e^n{}_i)^2$$

In order to acquire weights for different types of dependency edges, we have introduced finer granularity for edges which is based on the information provided by FEAT (Murphy et.al, 2001) and experimental scenarios. For instance, some methods only read object attributes and their weight is very small.

Experimental Results

We have used ranking entropy metrics to estimate the complexity of mobile agent application *MOB-Trader* written for mobile agent platform – aglets (Lange & Mitsuru Oshima, 1998). The application implements complex trading scenarios of multiple vendors (*Moderator* module) and mobile buyer agents (*Buyer* modules). In the *Buyer* modules, Aspect/J enhances the modularization by placing exceptions and handling of preconditions and post conditions in aspects instead of special classes and subclasses. Negotiation strategies range from Contract Net Protocol (CNP), Constraint Satisfaction Problem based solutions, and market based models (game theoretical models).

In order to provide implementation flexibility, each negotiation strategy specific rules and exception handling are implemented as a set of interchangeable aspects. We have also implemented the same application with and without aspects to measure reduction in entropy.

```
public aspect AspectForGetVendor{
```

```
    pointcut getListOfVendor(ListManager lMng, Message aMsg)
       :target(lMng)&&args(aMsg)&&call(public Hashtable ListManager.getVendor(..));

        before(ListManager lMng, Message aMsg)
            throws InvalidRegisKeyException, InvalidBusinessException : getListOfVendor(lMng,
aMsg){
               Hashtable buList = lMng.getBusinessList();
               Object infox  = aMsg.getArg("buType");
               String buType = infox.toString();
               System.out.println("Looking for->"+buType+" in Hash size   "+buList.size());

               if(buType==null) throw new InvalidRegisKeyException("Null key");
               if(!buList.containsKey(buType)) throw new InvalidBusinessException("No such
that business");
        }
}
```

Figure 1: Example of Aspect for Get Vendor

Aspect/J was used to enhance a modularisation by placing some particular exceptions that might occur and negotiation process in aspects instead of special modules or classes.

Figure 1 shows the example of code with aspects to handle two particular exceptions: '*InvalidRegis-KeyException*' and '*InvalidBusinessException*'. Before *getVendor()* method *of ListManager* class is called, the aspect named *AspectForGetVendor()* is executed to validate the register information: register key and business type. The aspect handles two exceptions *InvalidRegisterKeyException* and *Invalid-BusinessTypeException()*. The handling of each exception invokes additional submodule which provides customized handling.

```
public aspect AspectForStock implements Runnable{
    private Vector resList = null;
    private Thread resTimer = null;
    pointcut doQuotation(StockManager sMgr):target(sMgr)&&args(String)&&call
(public Quotation StockManager.queryItemPrice(..));
...some lines of code. . .
   after(StockManager sMgr) : doQuotation(sMngr){
        resTimer = new Thread(this);
        resTimer.start();
   }
   public void run(){
        resList   = sMgr.getStock();
        int index = (resList.size()-1);
        try { Thread.sleep(10*1000); …….
```

Figure 2: Aspect for dealing with threads

AspectForStock () aspect deals with negotiation mechanism implementation on multiple threads. The aspect is invoked after buyer agent has sent reservation message to seller host (Moderator module). It starts a thread (see Figure 2) to hold the reserved order for a while until it receives a payment notification message from buyer agent. If the response message is not received within acceptable period, the reserved order will be cancelled.

Protocol performance improvement can be achieved by replacing the relevant aspects with new aspect code. The other classes remain unchanged.

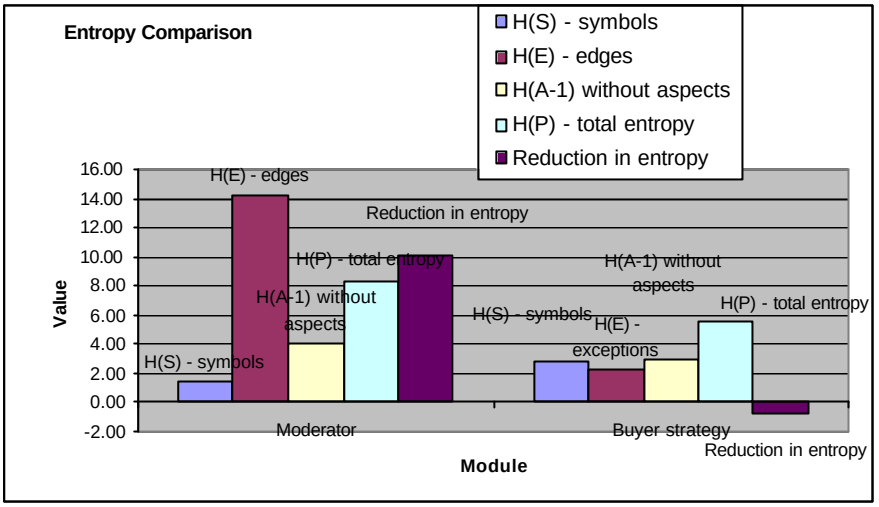

Figure 3: Entropy comparison

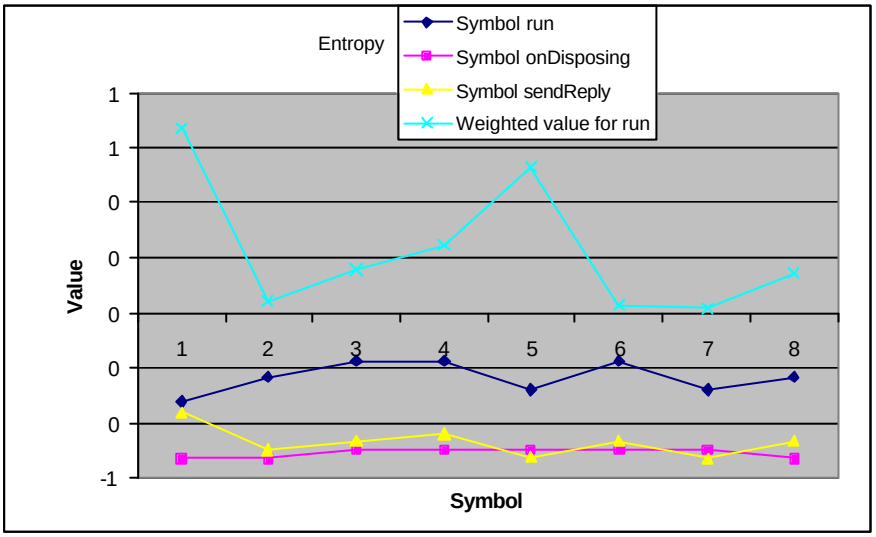

Figure 4Entropy values for selected symbols

Selected results of the experiment are in Figure 3 and Figure 4.

The *Moderator* module shows higher entropy for finding edge destination, path entropy (total entropy) and weighted entropy due to its multithreaded implementation. Entropy H(A-1) was calculated for both modules implemented without aspects.

The comparison in total entropy and weighted entropy is in Figure 5.

Weighted entropy shows smaller values in both modules because many methods are simple *"read"* type methods with very low weights.

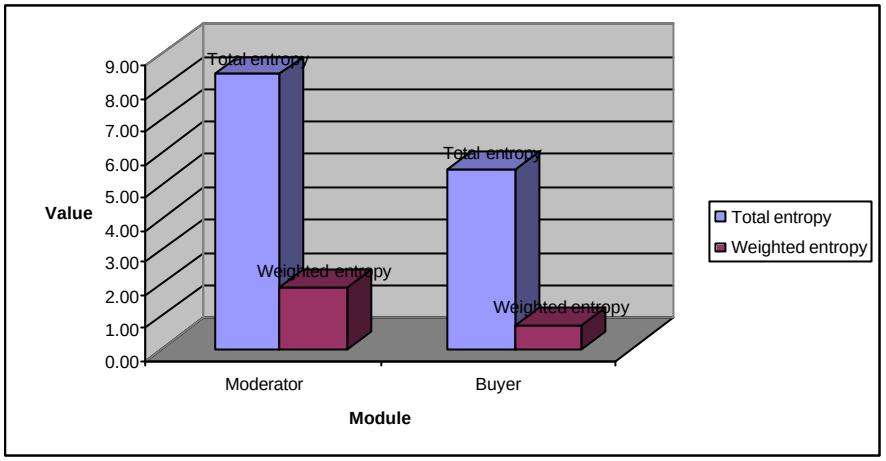

Figure 5: Comparison of total and weighted entropy

Usability Remarks on Metrics

This paper has provided an overview of entropy based metrics used in object oriented mobile agent development. The entropy metrics are useful in ranking different modules with regard to their complexity. Weighted entropy metrics provide means for assessing the complexity with regard to subjective edge type weight. Due to the limited space we have included only a few results from the experimental application *MOB-Trader* and we have omitted the methodology for collecting data and thorough explanation of results.

References

Aspect/J (2002). The AspectJ™ Programming Guide. Xerox Corporation, http://AspectJ/doc/progguide/printable.html.

Bansiya, J., Davis, C., Etzkorn, L. (1999). An entropy-based complexity measure for object-oriented designs. *Theory and Practice of Object Systems*, Vol. 5(2), pp.111-118.

Belady, L.A& Lehman, M.M. (1976). A Model of a large program development. *IBM Systems Journal*, Vol 15(3), pp.225-252.

Davis, J.S., & LeBlanc, R.J. (1988). A study of the applicability of complexity measures. *IEEE Transactions on Software Engineering*, Vol 14(9), pp.1366-1371.

Decker, K. & Lesser, V. (1995). *Analyzing the need for meta-level communication*. Computer Science Department. University of Massachusetts, Technical Report 93-22.

Dospisil, J. (2003). Software metrics, information and entropy. To appear in *Practicing Software Engineering in the 21st Century*. Editor: Dr. Joan Peckham.

Harrison, W. (1992). An entropy-based measure of software complexity. *IEEE Transactions of Software Engineering*, Vol. 18, No. 11, November, pp. 1025-1029.

Kendall, E. (1999). Role model designs and implementations with aspect oriented programming. *Proceedings of the 1999 Conference on Object- Oriented Programming Systems, Languages, and Applications* (OOPSLA'99), ACM Press, November.

Kephart, J. O., Hanson, J. E., Levine ,D. W., Grosof , B. N., Sairamesh, J., & Segal, R. B. (1998). Dynamics of an information-filtering economy . *Proceedings of Second International Workshop on Cooperative Information Agents (CIA-98)*. Paris.

Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J., Loingtier, M. & Irwin, J. (1997). *Aspect oriented programming*. Xerox Corporation. http://www.parc.xerox.com/spl/projects/aop/

Lange, D. B. & Oshima, M. (1998). *Programming and deploying Java mobile agents with Aglets*: Addison- Wesley.

Mass-Colell, Whinston A., R. & Green, J.R. (1995). *Microeconomic theory*: Oxford University Press.

Murphy, G., C., Lai, A., Walker, R.J. & Robillard. M.P. (2001). Separating features in source code: Exploratory study. *Proceedings of the 23th International Conference on Software Engineering*, Toronto, pp. 275-284.

Nareyek, A. (1998). Constraint-based agents. *Technical Report, German National Research Center for Information Technology*. Berlin.

Osborne, M.J. & Rubinstein, A. (1990). *Bargaining and markets*: Academic Press.

Sandholm, T.W. & Lesser, V.R. (1998). Issues in automated negotiation and electronic commerce: Extending the contract net protocol. *Readings in AGENTS* (ed. Michael N. Huhns &Munindar P. Singh), Morgan Kaufmann, 66-73.

Sen, S. & Durfee, E. (1994). The role of commitment in cooperative negotiation. *International Journal of Intelligent Cooperative Information Systems*, 3(1), 67-81.

Smith, R.G. (1980). The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers C-29(12)*:1104-1113.

Tarr, P., Ossher, H., Harrison, W, & Sutton, Jr. (1999). N degrees of separation: Multi-dimensional separation of concerns. *Proceedings of the 21st International Conference on Software Engineering.*