From AOP to MDSOC: An Experience Report

Christina von Flach G. Chavez* Alessandro Fabricio Garcia†
Carlos J.P. Lucena

e-mail: {flach, afgarcia, lucena}@inf.puc-rio.br

PUC-RioInf.MCC29/01 September, 2001

Abstract: In this report, we present some insights gathered through the implementation of the Portalware system using the IBM Hyper/J tool, having as a starting point an aspect-oriented design model and an aspect-oriented implementation (using the Xerox Parc's AspectJ tool). We present a preliminary mapping between the elements of the two approaches and discuss some alternatives for deriving an hyperslice-based solution from the Portalware aspect-oriented model. We describe the selected set of transformations we have applied, some potential problems and solutions related to the transformation process, and finally, we present the resulting Hyper/J solution.

One of the motivations behind this development exercise is a practical assessment of each approach's similarities and differences, strengths and weaknesses to help in the validation of a generic aspect-based design model that can be mapped to the implementation models supplied by AspectJ and Hyper/J.

Keywords: multi-dimensional separation of concerns, aspect-oriented programming, aspect-oriented modeling.

Resumo: Neste trabalho, apresentamos algumas lições aprendidas através da implementação do sistema Portalware usando a ferramenta Hyper/J da IBM, tendo como ponto de partida um modelo de design orientado a aspectos e uma implementação orientada a aspectos (em AspectJ, ferramenta da Xerox Parc). Apresentamos um mapeamento inicial entre os elementos das duas abordagens e discutimos algumas alternativas para a obtenção de uma solução baseada em *hyperslices* a partir do modelo orientado a aspectos do sistema Portalware. O conjunto de transformações utilizado é descrito, bem como alguns problemas e soluções associados ao processo de transformação. Por fim, apresentamos uma solução obtida em Hyper/J.

Uma das motivações subjacentes a este experimento é o entendimento de pontos em comum e diferenças, pontos fortes e fracos de cada uma das abordagens, para auxiliar na validação de um modelo de design baseado em aspectos que possa ser mapeado para os modelos de implementação de AspectJ e Hyper/J.

Palavras-chave: separação de concerns multi-dimensional, programação orientada a aspectos, modelagem orientada a aspectos.

*PhD Student at PUC-Rio and Lecturer at Federal University of Bahia (UFBA), Brazil.
†PhD Student at PUC-Rio.

1 Introduction

The identification and structuring of software components, crosscutting aspects, and their relationships are essential tasks in order to produce effectively reusable components and aspects. However, these tasks are not trivial and consequently require a disciplined design approach as well as suitable implementation tools. Aspect-oriented programming (AOP) and multi-dimensional separation of concerns (MDSoC) denote a set of approaches that provide new modularization and composition facilities to improve separation of concerns at the implementation level.

In this context, two research activities concerned to AOP and MDSoC are currently being held within the Software Engineering Laboratory (LES) at PUC-Rio: (i) the assessment and evaluation of applying aspect-oriented design and programming techniques to the design and implementation of multi-agent object-oriented software, and (ii) the assessment and evaluation of the strengths and weaknesses of approaches and tools that support AOP and MDSoC to help in the validation of a generic aspect-based design model under development.

An unifying subject of study for these activities is the development of Portalware [6], a web-based environment that applies groupware concepts in order to support a disciplined approach for the construction and management of e-commerce portals. Software agents were introduced in Portalware to assist its users with time-consuming activities and to automate repetitive user tasks. An innovative aspect-based agent model proposed in [7] was applied in the early design of the Portalware system, to allow the separation and flexible integration of multiple aspects of agenthood, such as interaction, autonomy, adaptation and others. Moreover, an exploratory implementation for Portalware was developed using AspectJ [14], an aspect-oriented extension to Java. Some interesting results have been found and discussed elsewhere [8, 10].

The resulting aspect-based design model (highly influenced by the AspectJ programming model) and the AspectJ/standard Java code were reused in the implementation of Portalware with Hyper/J [18], a tool that supports MDSoC in standard Java software. The adoption of an aspect-based design model supports our belief that the base-aspect dichotomy simplifies concern-based modularity [17].

The purpose of this work is to present the main steps related to the reimplementation of Portalware and to report some insights we had in implementing the same application, the Portalware system, using both AspectJ and Hyper/J, having as a starting point an aspect-oriented design model. One of the motivations behind this development exercise is the assessment of each approach's strengths and drawbacks to help in the validation of a generic aspect model under development that can be mapped to the implementation models supplied by AspectJ and Hyper/J.

In section 2 we present some background information, including an overview to AspectJ and Hyper/J, and a brief introduction to the Portalware multi-agent system. In section 3 we describe the aspect-based model adopted in the design of the Portalware system. In section 4, we describe a mapping from the aspect-based design model and aspect-oriented implementation (expressed in AspectJ) to Hyper/J. In section 5, we present some insights we have gathered during the transformation process.

2 Background

2.1 Separation of Concerns

AOP and MDSoC denote a set of approaches that provide new modularization and composition facilities to improve separation of concerns (SoC) at the implementation level. Usually, each new approach prescribes its own programming model to give support to SoC. AspectJ [14] and Hyper/J [18] are two evolving tools, recognized as the main representatives for AOP and MDSoC, respectively.

2.1.1 AspectJ

AspectJ [14] is an aspect-oriented extension to Java developed at Xerox Parc [2]. In the AspectJ model, **components** refer to classes and objects, and new programming elements are added to Java to reify the AOP core concepts:

- **Aspects**: An *aspect* is a new kind of module used to localize crosscutting implementation. Aspects comprise pointcut, advice (which provide the crosscutting implementation) and class-like member definitions.

  The AspectJ's implementation model defines aspects as types and aspect instances as regular objects that can not be explicitly instantiated. In general, an aspect instance is defined as a member of the object it crosscuts, i.e., objects may embody the crosscutting relationship and aspect instances inside their structure.

- **Join points** and **pointcuts**: The current version of AspectJ adopts a model in which a *join point* is a principled point in the dynamic execution[1] of a Java program [14], corresponding to method/constructor calls, method/constructor receptions, field accesses, implicit exception handler calls, etc.

  A *pointcut* is a description of set of join points and of some values in their execution context.

- **Advices** and **introductions**: An *advice* comprises aspect behavior that is supposed to be executed at each join point in a pointcut. Advices are defined by *before*, *after*, and *around* declarations. These declarations associate the aspect code with a pointcut and a time, indicating how the aspect code will be combined with the base code.

  An *introduction* defines attributes and operations to be added to the base code.

- **Crosscutting**: Crosscutting is defined over class operations and fields. AspectJ promotes *operation-level composition* [16], through the specification of join points as class and method signatures that the aspect is supposed to crosscut (by explicit enumeration or using wildcard mechanisms).

- **Weaving**: Up to the current version of AspectJ (0.8b2), the weaving process corresponds to a pre-processing step at compile-time.

- **Precedence**: AspectJ provides rules to establish the relative precedence among aspects, for situations when more than one advice applies at a join point (e.g., the *dominates* relationship).

[1] In AspectJ's introduction mechanism the join points are types and classes.

AspectJ source code is divided into interface, class and aspect modules. The AspectJ compiler (`ajc`) takes a set of interfaces, aspects and classes, and coordinates the crosscutting of the aspects and the compilation into class files or (optionally) the generation of Java source files. For a complete overview of these elements and other AspectJ features, see [14].

2.1.2 Hyper/J

Hyper/J [18] is a tool developed at IBM Watson Research Center to support Hyperspaces [12], an evolution on subject-oriented programming [11]. Hyper/J supports the identification, encapsulation and integration of concerns from standard Java programs, without requiring special language extensions, coding conventions or packaging.

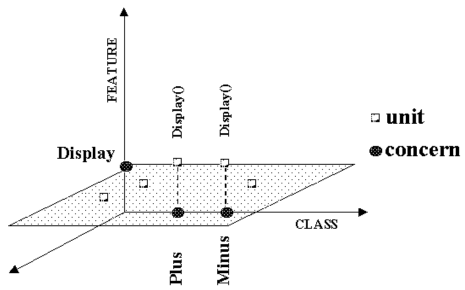

Figure 1: Hyperslice.

In Hyperspaces, an **hyperspace** is defined as a set of software units that compose a software system, organized in a multi-dimensional matrix, where each axis represent a *dimension of concern*, each point on an axis represents a *concern in that dimension*, and the coordinates of a software unit indicate all the concerns it affects. Each dimension of concern can be viewed as a partition of the set of units: one particular software decomposition.

The main elements of Hyperspaces supported by Hyper/J are:

- **Hyperslices**: An *hyperslice* is a set of units (packages, classes, methods, fields, etc.) that comprises a concern; hyperslices are declaratively complete, that is, they must declare everything to which they refer. An hyperslice generalizes the concept of *subject* from subject-oriented programming [11]. Figure 1 presents a graphical view of an hyperslice. This hyperslice consists of 5 units that compose a Display feature in the Feature dimension (vertical axis).

- **Corresponding Units**: Corresponding units are units (packages, classes, methods, fields, etc.) that belong to different hyperslices and that match each other according to some criteria. Corresponding units are supposed to be composed (with merge or override integration relationships) to form a new unit of software that contains some or all of the functionality of the original units.

- **Composition Strategy**: The specification of integration relationships in Hyper/J follows an approach where developers first specify a general strategy for identifying matching units across hyperslices, and then defining exceptions to, or specializations of, that strategy for those cases where the strategy does not apply. Hy-

	AspectJ	Hyper/J
Crosscutting concern	aspect	hyperslice
Composition where how	 join point merge before, after, around	 corresponding unit merge, override
Composition Specification	inside aspect	outside hyperslice
Composition Precedence	dominates implicit rules	order hyperslice declaration
Composition Time	compile-time	compile-time

Table 1: Comparing AspectJ and Hyper/J

per/J supports three general strategies: *mergeByName*, *nonCorrespondingMerge*, and *overrideByName*.

- **Precedence**: Hyper/J provides rules to establish the relative precedence among composing hyperslices. The *order* relationship indicates that the order of related units is significant, and describes any ordering constraints.

Before using the Hyper/J tool on a set of java classes, the developer must provide three inputs: an hyperspace file, one or more concern mapping file(s) and an hypermodule file. The **hyperspace file** is similar to a project description. It lists all of the Java class files with which a developer is working, and to which the developer wishes to apply Hyper/J. The **concern mapping file(s)** describe how various parts of Java class files address different concerns in an hyperspace. For example, a developer may declare that any method named `display-something` addresses the Display concern. The **hypermodule file** describes which hyperslices should be integrated and how the integration should proceed, including the general composition strategy, *equate* clauses, *order* clauses, etc.

Once the files described above have been written, the Hyper/J compositor tool can be applied to produce a new set of integrated units – a new composed hyperslice.

2.1.3 Comparison

The AspectJ programming model supports the base-aspect dichotomy, that is, it distinguishes between the basic concern – the *base* – and crosscutting concerns – modularized by *aspects* – that affect the base behavior. Composition between base and aspects is defined in terms of base-related *join points* [14]. Crosscutting behavior can be added *before*, *after* or *around* join points. Composition is defined inside aspects. Precedence among aspects is resolved implicitly (before, after, around rules) or explicitly (the *dominates* clause).

Hyper/J supports Hyperspaces [18], an evolution on the early work on subject-oriented programming (SOP) [11] that does not necessarily distinguish between base and crosscutting concerns. Concerns are modularized using *hyperslices*. Composition rules are defined in terms of *corresponding units* [18]. These units are related by *merge* or *override* relationships. Composition is defined independently from the hyperslices. Precedence among hyperslices is resolved explicitly (declaration of hyperslices in the hypermodule, the *order* clause). Table 1 summarizes some of the characteristics of AspectJ and Hyper/J.

Agency Property	Definition
Interaction	An agent communicates with the environment and other agents by means of sensors and effectors
Adaptation	An agent adapts/modifies its mental state according to messages received from the environment
Autonomy	An agent is capable of acting without direct external intervention; it has its own control thread and can accept or refuse a request message
Learning	An agent can learn based on previous experience while reacting and interacting with its environment
Mobility	An agent can move from one environment in a network to another
Collaboration	An agent can cooperate with other agents in order to achieve its goals and the system's goals

Table 2: An Overview of Agency Properties [7]

2.2 Portalware

Portalware [6] is a web-based environment that applies groupware concepts in order to support a disciplined approach for the construction and management of e-commerce portals.

Each Portalware user can be assigned one or more responsibilities: the main ones are (i) content supplier, (ii) reviewer, and (iii) editor. An user plays the role of *content supplier* when he/she is responsible for providing the portal with information (for instance, news). A *reviewer* is an user that examines each of the content segments, and may change them. The editor is responsible for selecting from the available content segments for publishing and for inviting individuals to be reviewers or content suppliers. Software agents have been introduced in Portalware to assist its users with time-consuming activities and to automate repetitive user tasks.

2.2.1 Agenthood

The definition of agenthood adopted by Portalware can be found in [7] and is summarized here. *Software agents* are often viewed as complex objects with an attitude, in the sense of being objects with some additional properties. An agent is generally regarded to be an autonomous entity that has capabilities, interacts with its environment, and adapts its state and behavior based on this interaction. The state of an agent is formalized by knowledge, and is expressed by mental components such as beliefs, goals, and plans. The behavior of an agent is driven by plans. Each software agent typically has different non-functional properties and collaborative capabilities, that affect agent's state and behavior.

Non-functional Properties. Each software agent may have different non-functional properties. Table 2 presents a brief definition for the main non-functional properties of interest in the agent domain.

Collaborative Capabilities. An agent can use the capabilities provided by other agents via some communication language. Collaborative capabilites are application-dependent and are specific for each context. So, since software agents can cooperate

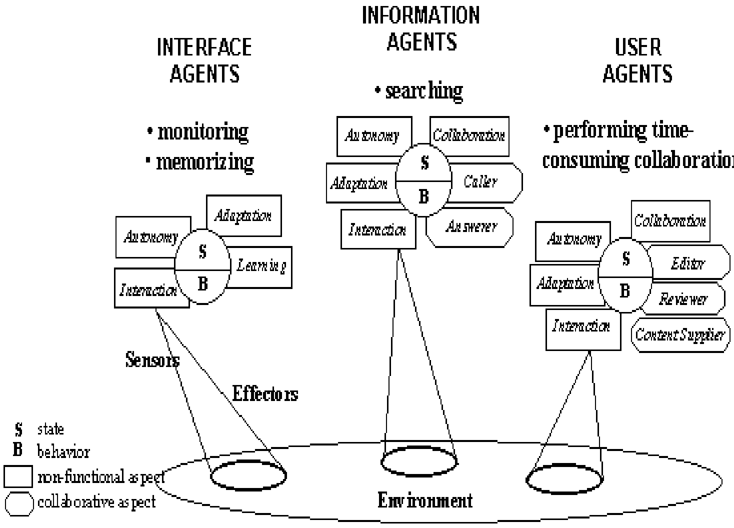

Figure 2: Portalware Agents [7]

while pursuing their goals in different situations, a cooperating agent generally includes different collaborative capabilities in order to work together in multiple contexts. In order to perform a cooperation, a collaborative plan is instantiated, and it chooses the eligible collaborative capabilities.

Interacting and Overlapping Properties. In general, non-functional properties are not ortoghonal they interact with each other. For instance, the adaptation depends on autonomy since it is necessary to adapt the agents state (beliefs and goals) and behavior (plans) when the autonomy property decides accepting an incoming message. In addition, two properties are overlapping: interaction and collaboration. Collaboration is viewed as a more sophisticated kind of interaction, since the former comprises communication and coordination. Interaction is only concerned with communication, i.e., sending and receiving messages. During a collaboration, messages are also received from and sent to the participating agents. However, the collaboration property additionaly defines how to collaborate, and addresses some coordination protocol [10].

2.2.2 Agent Types

Portalware encompasses three agent types with different capabilities (Figure 2): (i) interface agents, (ii) information agents, and (iii) user agents. *Interface agents* monitor the graphical interface in order to interact with Portalware users. They may learn short-cuts by capturing preferences, and by receiving explicit instructions from the user. *User agents* represent Portalware users and are implemented to reduce the need for cross-talk between working users. Since editors, reviewers and content suppliers need to communicate with each other to maintain the web portal, user agents incorporate collaborative capabilities for automating and supporting collaboration in different contexts. Finally, Portalware users often need to search for information which is stored into different databases. *Information agents* are attached to databases, and contain plans to search for information.

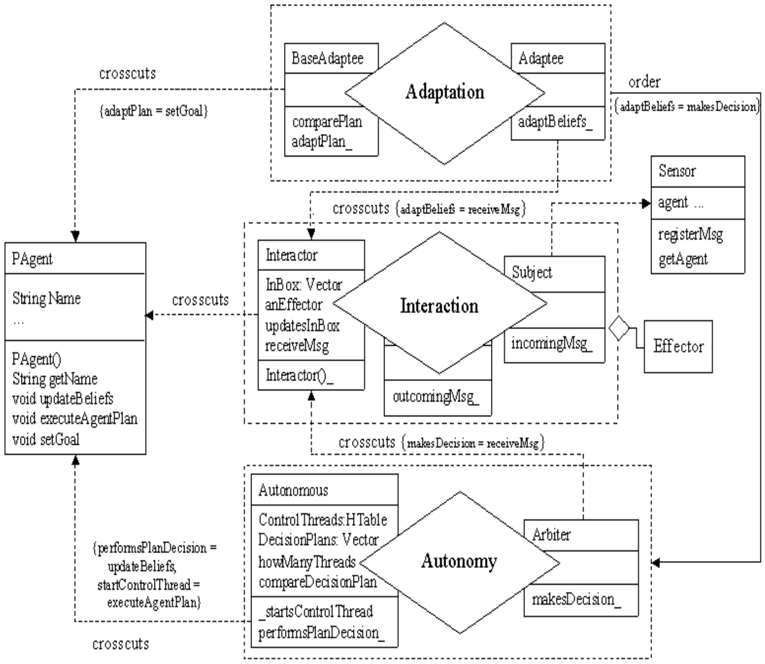

Figure 3: An Aspect Model for Portalware Agents.

3 An Aspect-based Model for Portalware

To deal with the intrinsic complexity on engineering a multi-agent system like Portalware, an aspect-based approach that provides separation of concerns among the different non-functional properties and collaborative capabilities has been proposed [7, 8]. Classes and inheritance are used to model the core state and behavior of agents, as well as different types of agents. Aspects are used to model properties and capabilities to be combined (or weaved) to these classes. This approach allows the composition of properties in a disciplined and non-intrusive manner. As a consequence, we claim it supports the construction of multi-agent software with improved structuring for evolution and reuse of design and implementation solutions. We are currently developing a comparative study to validate this hypothesis [9].

Figure 3 illustrates part of the aspect model used in the implementation of Portalware. Collaboration capabilities are not addressed in this report. In particular, the figure shows the definition of aspects, pointcuts and introductions. The three aspects presented interact. An ordering relationship between the `Adaptation` and `Autonomy` aspects must be explicitly established, since they advise the same join point. The notation used [3] is briefly explained in Figure 4.

3.1 An AspectJ Solution

We have developed an exploratory implementation for Portalware using AspectJ. It addresses three basic non-functional properties (interaction, autonomy and adaptation) as well as other properties not described here (collaboration among information agents and

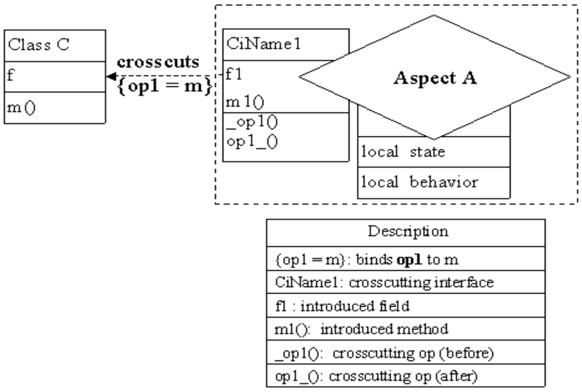

Figure 4: The structural facet of the aspect model.

collaboration among user agents).

Aspects are used to implement agent's non-functional properties. The three aspects presented in Figure 3 interact and, in order to implement the precedence relationship between the `Adaptation` and `Autonomy` aspects, the "dominates" relationship of AspectJ is used. Classes are used to implement agent's functional properties (related to its beliefs, goals, plans, etc.). The class `PAgent` and its subclasses (`InformationAgent`, `UserAgent`, `InterfaceAgent`) are the base classes that aspects crosscut.

Figure 5 illustrates the `Interaction` aspect. This aspect introduces `effector`, `inBox`, `outBox`, and related methods in `PAgent`, to provide communication with the environment and other agents. Two pointcuts are defined: the first one provides means to initialize the new members introduced in `PAgent` after instance creation, and the second pointcut defines actions to be taken after any agent sends a message.

For a software agent to be autonomous, it must have a separate process or thread. Figure 6 illustrates part of the `Autonomy` aspect. The `Autonomy` aspect introduces two fields (`ControlThreads` and `DecisionPlans`) and two methods (`howManyThreads` and `compareDecisionPlan`) in `PAgent` class, and defines three pointcuts and three advices over methods defined or introduced in `PAgent`.

4 From Aspects to Hyperslices

The design model induced by the AspectJ programming model, as well as the source files (containing AspectJ code and standard Java code) were provided as input to the design of an MDSoC solution, implemented with Hyper/J.

The transformation between the AOP model and the MDSoC model requires the mapping between their elements (see again Table 1). Hyper/J offers at least two ways to configure Portalware from the AspectJ source files. We discuss the two alternatives here but we only elaborate on the one that has been selected (see 4.2).

4.1 Two Alternatives

We use part of the `Autonomy` aspect to illustrate two possible alternatives for the transformation from the AOP model to the MDSoC model. The `Autonomy` aspect defines a pointcut `makesDecision` that crosscuts the method `receiveMsg`.

```
aspect Interaction of eachobject(instanceof(PAgent)) {

    public Vector PAgent.InBox;                 public Vector PAgent.OutBox;
    public Effector PAgent.AgentEffector;       public Sensor PAgent.AgentSensor;

    public void PAgent.receiveMsg(Message msg) { ... }    public void PAgent.sendMsg(Message msg) { ... }
    public void PAgent.updatesInBox(Message msg) {...}    public void PAgent.updatesOutBox(Message msg)  {...}
    public void PAgent.addEffector(Effector effector) { ... }   public void PAgent.addSensor(Sensor sensor) { ... }
      ...
    public void Pagent.initial(PAgent agent) {
        PAgent.InBox = new Vector();  PAgent.OutBox = new Vector();
        Sensor sensor1 = new Sensor(agent.getName(), agent);    Effector effector1 = new Effector();
        agent.addSensor(sensor1);    agent.addEffector(effector1);
    }
1   after(PAgent pa) returning (): instanceof(pa) && receptions(new(..)) {
        System.out.println("<* Interaction *> creating sensor and effector.");
        pa.initial(pa);
    }
2   pointcut outcomingMsg(PAgent sender, Message msg) :  instanceof(sender) && executions(void send*(msg));

    after(PAgent sender, Message msg): incomingMsg(sender, msg) {
       System.out.println("<* Interaction *> Sending the message: " + msg);
       sender.updatesOutBox(msg);
       Effector effector = sender.getEffector();     effector.sendMsgBlackboard(msg);   }
}
```

Figure 5: The Interaction Aspect.

```
aspect Autonomy of eachobject(instanceof(PAgent)) {

     pointcut makesDecision(PAgent receiver, Message msg):
          (instanceof(receiver) && executions(void receiveMsg(msg)));

     after(PAgent receiver, Message msg): makesDecision(receiver, msg)
     { advice body }
}
```

In the first alternative – which we call the **mergeByName** alternative – the method name `receiveMsg` is used to name the body of advice that uses the pointcut `makesDecision`. The aspect name is used to name a new package, and the class name affected by the aspect is used to name a new class created in this package. This alternative is simpler to implement and generates a smaller hypermodule file.

```
package Autonomy;
class PAgent {
   public void receiveMsg { advice body }
}
//  In the hypermodule file:

relationships:  mergeByName;
```

However, if wildcards are used within the primitive pointcut (for example, send*), depending on the number of method names that match the pattern, this solution may

```
aspect Autonomy of eachobject(instanceof(PAgent)) {

   private Hashtable PAgent.ControlThreads = new Hashtable();
   private Vector PAgent.DecisionPlans = new Vector();

   public int PAgent.howManyThreads() {...}
   public synchronized boolean PAgent.compareDecisionPlan(Proposal prop, Plan plan) {...}

1  pointcut performsPlanDecision(PAgent pa):   (instanceof(pa) && executions(void updateBeliefs(Message)));

   after(PAgent pa): performsPlanDecision(pa) {
     System.out.println("<* Autonomy *> Decides if due to the update of beliefs, a plan must be performed.");  ... }

2  pointcut startsControlThread(PAgent pa, Plan plan, Goal goal):
          (instanceof(pa) && executions(void executeAgentPlan(plan, goal)));

   before(PAgent pa, Plan plan, Goal goal): startsControlThread(pa, plan, goal) {
     System.out.println("<* Autonomy *> Starts control thread - starts execution of the plan.");
     plan.executePlan(pa, goal);   }

3  pointcut makesDecision(PAgent receiver, Message msg):
                                  (instanceof(receiver) && executions(void receiveMsg(msg)));

   after(PAgent receiver, Message msg): makesDecision(receiver, msg) {
   System.out.println("<* Autonomy *> After receive  : Verifying if it is a proposal msg: " +
                  msg + " from: " + msg.getSender());      ... }

}
```

Figure 6: The Autonomy Aspect.

lead to code redundancy. We believe it is better suitable for domain-specific aspects (such as the three Portalware properties /aspects presented here) that possibly do not need to use wildcards.

In the second alternative – which we call the **equate and mergeByName** alternative – the pointcut name `makesDecision` is used to name the advice body. The new generated method must be explicitly *equated* with to the base method `receiveMsg`. The aspect name is used to name the new package and also the new class created in this package.

```
package Autonomy;
class Autonomy {
   public void makesDecision { advice body }
}
//  In the hypermodule file:

relationships:  mergeByName;

equate operation Interaction.Kernel.receiveMsg,
                 Autonomy.Kernel.makesDecision,
                 Adaptation.Kernel.adaptBeliefs
   into Agent.Kernel.receiveMsg;    ...
```

This alternative improves code readability, since the AspectJ pointcut names are kept (transformed into method names) and the corresponding units are declared explicitly, but the hypermodule file may become full of `equate` declarations. Besides, this alternative is possibly better in the presence of a heavy use of wildcards – probably in the context of canonical aspects such as exception handling, persistence, etc., when applied to a wide range of classes.

4.2 The Transformation

We have adopted the **mergeByName** alternative, which requires a simple set of transformation rules, and possibly provides a smaller hypermodule file. This solution is based on (i) rewriting each aspect into one or more classes encapsulated by a separate package, (ii) transforming advice code into ordinary method code and (iii) adopting the *mergeByName* general composition strategy.

4.2.1 Initial Conditions

The following initial condition held, simplifying the transformation process:

- The Java source files were available and compilable[2]
- Each aspect/class was defined in a separate file.
- Every pointcut used only the "executions" primitive pointcut.
- Pointcuts did not make extensive use of wildcard mechanisms.
- For each aspect definition, no before/after advices were defined over the same pointcut and there were no around advices.

4.2.2 A Set of Rules

A set of transformation rules were applied following the steps described below:

1. Separate files with standard Java code from files with AspectJ code[3].
2. Create a new file .hs (hyperspace file), declaring the class names that will live in the hyperspace.
3. Create a new file .hm (hypermodule file).
4. For each aspect $A_i, i = 1, n$ and $j \neq i$ with header

   `aspect` A_i `[dominates` A_j `] of eachobject(instanceof(CName))`

   (a) Create a package named A_i

   (b) Create a file named A_i/concerns.cm containing: `package` A_i `:` A_i`.Kernel`
5. Declare in the file .hm

   (a) The "base" hyperslice

   (b) The hyperslices A_i.Kernel, for each aspect $A_i, i = 1, n$ and $j \neq i$ containing

   `aspect` A_i `[dominates` A_j `] of eachobject(instanceof(CName))`

   where, if `Ai dominates Aj`, then A_i.Kernel is declared after A_j.Kernel[4].

[2] Nevertheless, Hyper/J does not require the source files, only class files.
[3] The original files with standard Java were not rewritten.
[4] For overlapping through after advices.

(c) The general composition strategy: `mergeByName`

6. For each aspect $A_i, i = 1, n$ and $j \neq i$ such that

   `aspect` A_i `[dominates` A_j `] of eachobject(instanceof(CName))`

   (a) Create a file named A_i/`Cname`.java and declare class `Cname` in it

   (b) Copy introductions from aspect A_i to class `Cname`

   (c) For each pointcut that contains `executions(T methodName)`,

      i. Define new method methodName with body B defined in the advice

      ii. If advice defined on pointcut is `before`
      Declare in the file .hm

      ```
      order  action Ai.Kernel.CName.methodName
             before action Agent.Kernel.CName.methodName;
      ```

      iii. If T $\neq$ void, create a summary function and declare `set summary function` in the file .hm.

Figure 7 illustrates the class `PAgent`, generated in the new package `Adaptation`. The same process will be applied to the other non-functional properties that crosscut the class `PAgent`, deriving other packages with new classes named `PAgent`. These classes will be merged by the Hyper/J tool.

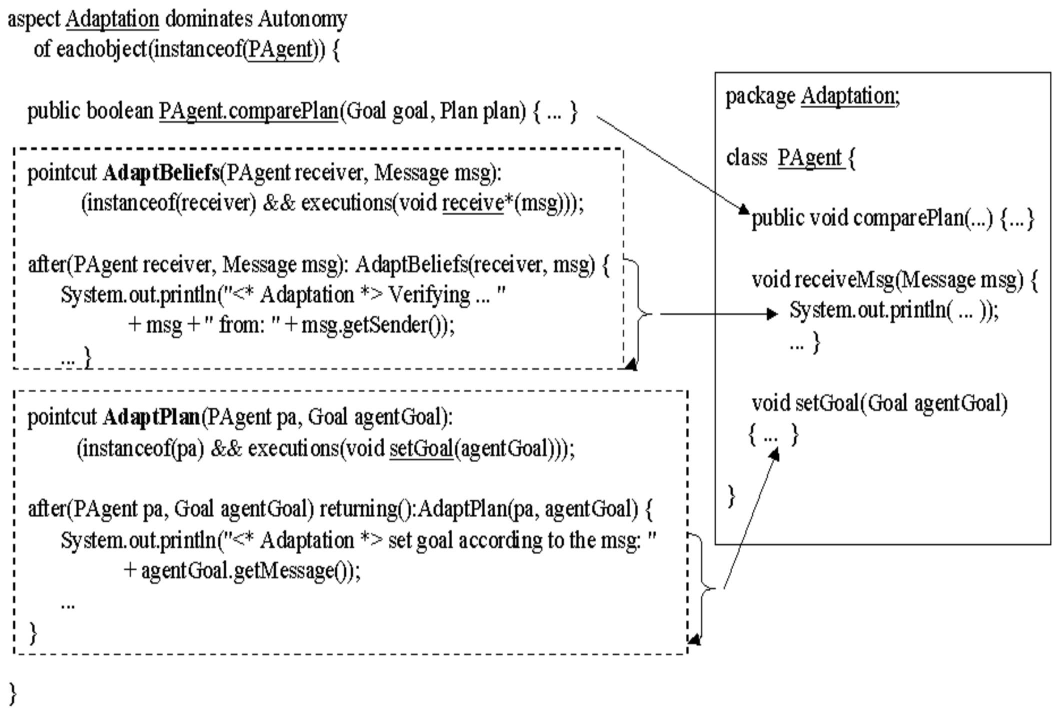

Figure 7: From the Adaptation aspect to the Adaptation hyperslice

4.3 An Hyper/J Solution

The resulting solution in Hyper/J was structured in an hyperspace with four dimensions (Agent, Interaction, Autonomy, Adaptation), four main hyperslices (Agent.Kernel, Interaction.Kernel, Autonomy.Kernel, Adaptation.Kernel), an hyperspace file, a concern map file for each dimension, an hypermodule file, and a set of standard Java classes. Each source aspect file with an AspectJ aspect declaration has generated one or more Java files, depending on the requirements of declarative completeness imposed by Hyper/J.

In the hyperspace file (figure 8), the Java class files with which a developer is working are listed. The classes that belong to the last three packages were generated from the AspectJ source files, after applying the transformation described in 4.2.

```
hyperspace Portalware
   composable class Agent.*;

   composable class Interaction.*;
   composable class Autonomy.*;
   composable class Adaptation.*;
   ...
```

Figure 8: Hyperspace file

A concern map file is created for each dimension (figure 9). The developer encapsulates the non-functional properties as first-class concerns by creating a new dimensions and describing how existing units in the hyperspace address concerns in each dimension. For example, the units in the package `Interaction` address the `Interaction.Kernel` concern in the Interaction dimension.

```
package Agent : Agent.Kernel
...
package Interaction : Interaction.Kernel
...
package Adaptation : Adaptation.Kernel
...
package Autonomy : Autonomy.Kernel
...
```

Figure 9: Concern map file

The hypermodule file (figure 10) describes that four hyperslices (`Agent.Kernel`, `Interaction.Kernel`, `Autonomy.Kernel` and `Adaptation.Kernel`) should be integrated with the `mergeByName` general composition strategy, and one exception in the ordering of hyperslice composition (the action `executePlan` from `Autonomy.Kernel` should be executed before the action `executePlan` from `Agent.Kernel`).

This solution has been derived using the transformation steps described above; it corresponds to a preliminary running implementation for Portalware using Hyper/J.

```
hypermodule Portalware
   hyperslices:
      Agent.Kernel,
      Interaction.Kernel,
      Autonomy.Kernel,
      Adaptation.Kernel;

   relationships:
      mergeByName;

      order  action Autonomy.Kernel.Plan.executePlan
      before action Agent.Kernel.Plan.executePlan;
end hypermodule;
```

Figure 10: Hypermodule file

5 Discussion

5.1 Problems and Possible Solutions

The transformation from the AOP model reified by AspectJ to the MDSoC model reified by Hyperspaces may require a lot of additional work. When the initial conditions reported in section 4.2 do not hold, some kind of refactoring in some methods inside the aspect definition. We assume as a pre-condition that the standard Java code (the base code) should not be refactored nor invasively modified. Below, we have listed some problems and possible solutions related to the transformation between the two models.

1. Problem: Two or more before/after advice are defined over the same pointcut (over the same method M) in the same aspect.

   Solution: Rename the derived method names with integer suffixes, sequentially (e.g., M1, M2, etc.). Define an *equate* clause to relate them to the original method M, using the name of M after the keyword `into`. Since Hyper/J can, by default, choose to run the merged code for the original methods in any order, an *order* clause must also be explicitly defined.

   ```
   package otherPackage;
   class ClassName {
      public void M { original body }
   }

   package AspectName;
   class ClassName {
      public void M1 { before-advice body 1 }
      public void M2 { after-advice body 2 }
   }
   //  In the hypermodule file:

   relationships:  mergeByName;

   order  M1, M, M2;
   equate M1, M, M2 into M;
   ```

2. Problem: Introduce and advice over the introduced method defined in the same aspect.

   Solution: Rename the advice (as described before) and equate the introduced operation and the renamed operation, using the name of the introduced operation after the keyword `into`. Define *order* clause that reflects the desired precedence between methods.

3. Problem: Around advice.

   Solution: Clone the original method M, naming the clone as OP-M. Generate a new method named M from the advice body and replace the *proceed* statement (from AspectJ) with an explicit call to OP-M. Use *override* integration relationship explicitly (instead of *merge*), relating M and OP-M.

```
package otherPackage;
class ClassName {
   public void M { original body }
}

package AspectName;
class ClassName {
   public void OP-M { original body from otherPackage }
   public void M { around-advice body }
}
//  In the hypermodule file:

relationships:  mergeByName;

override action SomeDimension.otherPackage.ClassName.M with
         action SomeOtherDimension.AspectName.ClassName.M;
```

4. Problem: The base operation and advices return values.

   Solution: Write a summary function that deals with multiple values to be returned and declare a `set summary function` in the hypermodule file. This may not be trivial and we are still working on this subject.

We had to define many units in some hyperslices, mainly methods with empty bodies, just for the sake of declarative completeness required by Hyper/J. One simple solution to the problem of discarding the units the return values is simply to map them to a `None` default concern in the concern mapping file.

Other issues are the handling of wildcards in pointcuts and the special variable `thisJoinPoint` from AspectJ. As we have previously discussed (section 4.1), the alternative **equate and mergeByName** provides a better solution to the translation of wildcards, but in the presence of the `thisJoinPoint` variable, unless we use reflection from Java, the **mergeByName** alternative must be used. This decision depends highly on the number of methods matched by the wildcard.

5.2 Some Insights

About the base-aspect dichotomy. Skipper [17] compares subject-oriented programming and aspect-oriented programming and drops interesting conclusions shared by this development experiment. For example, the base-aspect dichotomy, with explicit dependency on some "base" whose vocabulary is shared among the "aspects", may be regarded as a discipline that may simplify the development of MDSoC solutions, especially the rules of composition.

About the transformation. Lai, Murphy and Walker [15] describe an experiment with Hyper/J and discuss some code restructuring used to enable the capturing and composition of concerns. In this experiment, we have shared both the reported benefits and limitations presented ([15], section 4.2), although we did not restructure classes; instead, we have rewritten aspects into one or more classes. During the transformation process, we have noticed that AspectJ's aspects and named pointcuts are good mechanisms for improving the readability of the crosscutting code. By restructuring each aspect definition into multiple classes, rewriting advice code into ordinary methods, and fully using the *mergeByName* composition strategy in Hyper/J, we have simplified the transformation process, but we may have lost some of the reported benefits provided by aspects and named pointcuts [1]. Besides, sometimes the expected simplification provided by the *mergeByName* strategy is not achieved due to a very usual practice identified in AspectJ programs: the addition of a method definition to the base via an introduction and a subsequent advice defined on it, all inside the same aspect(Problem 2, section 5.1). Since we can not have two methods with the same signature in a class, this would require a more complex rewrite rule, with different naming convention for the advice and the inclusion of an additonal *equate* clause in the hypermodule file.

About AspectJ's join point model. Although we believe that the adoption of an aspect-based design model simplifies concern-based modularity, AspectJ's join point model provides at the implementation level much more than we are currently able to describe elegantly at the design level. For this reason, the preliminary version of the aspect model presented in 3 adopts a simple operation-level join point model. Fortunately, our initial approach to modeling Portalware could be easily reified by AspectJ, and the primitive pointcut `executions` has been used in order to express the join points of interest.

About the composition mechanisms. The experiment also allowed us to evaluate how the composition mechanisms of each tool work in practice. In AspectJ, even with the expressive power of wildcards[5] in pointcuts, it may be eventually necessary to invasively modify an aspect definition, because the composition between aspects and classes is hardwired in the aspect header. Moreover, the use of *dominates* establishes a strict relationship among aspects, and consequently, among *all* their pointcuts. It would be interesting if the designer could specify exceptions to the composition order for some of the pointcuts. Hyper/J provides a more elegant and flexible solution, by requiring the definition of the composition strategy explicitly outside the hyperslice definition and by allowing the specification of exceptions to the composition strategy and to the composition order among units (also at the level of operations). This approach increases

[5]In fact, we believe that the use of wildcards decreases system robustness and increases indirect coupling among classes and aspects.

the potential for non-invasive modification and reconfiguration. Nevertheless, current limitations of the composition mechanisms of both tools have hampered the achievement of an important requirement stated for `Portalware`: the dynamic attachment of different roles to distinct instances of agents [7]. Aspects are used to represent the different collaborative capabilities (roles) an agent can play. Kendall [13] describes the application of AOP to implement role models. Some of her guidelines were followed in the definition of the aspect-based model [8] (related to the implementation of agent's collaborative aspects). However, Kendall used an old version of AspectJ, where aspects could be dinamically attached to components – this feature is not supported by AspectJ anymore.

5.3 AspectJ or Hyper/J?

After an experiment involving two approaches that supports separation of concerns with different characteristics, and the definition of mappings between their concepts, a discussion about the strengths and drawbacks of each approach, under different perspectives, would be an expected contribution. So, in the remainder of this section, we present a comparison between AspectJ and Hyper/J that may be useful as an initial assessment for choosing between them.

- Goals

  Both approaches share a motivation on improving software modularity at the implementation level. AspectJ provides clean modularization of crosscutting concerns through the use of a new unit called "aspect", while Hyper/J supports Hyperspaces, and uses hyperslices to modularize concerns. Furthermore, Hyperspaces aims to support explicit identification of dimensions and concerns at any stage of software development.

- Requirements

  AspectJ requires the availability of standard Java files and generates standard Java class files – this is referred as *white-box AOP*. Hyper/J works on and generates standard Java class files – this is referred as *class migration*.

- Language Extension

  AspectJ extends Java with special-purpose language mechanisms and a richer join point model. Hyper/J does not require special coding conventions or packaging.

- Composition Mechanisms

  In AspectJ, composition is defined in terms of join points. AspectJ join points are principled points in the dynamic execution of the program. AspectJ provides a powerful, expressive syntax to define these points. In Hyper/J, composition is defined in terms of correponding units (class, interface, operation, action, field), i.e., join points are static and refer to points in the source code of the program.

  Other important characteristics have been discussed in 5.2.

- Support

  AspectJ is supported by Xerox Parc. AspectJ's team provides intensive support (including an active users mailing list and web page), and AJDE (AspectJ Development Environment), an environment that has been integrated with many Java development environments. AspectJ 1.0alpha1 has been released in 08/16/2001. Hyper/J is supported by IBM. Hyper/J's user list remains in absolute silence for almost a year. IBM's researchers involved with this project aim to give support to Hyperspaces at the design phase (Hyper/UML?) and later, at the requirements phase, providing alignment among requirements, design and code throughout the software life cycle. Hyper/J is available at IBM's alphaWorks site (last update: 03/30/2001).

6 Conclusions

We have reported some insights collected during the implementation of the same application, the `Portalware` system, using AspectJ and Hyper/J tools, having as a starting point an aspect-oriented design model. These insights refer not to the maturity or performance of each tool, but to their ability to separate and compose concerns, and the correspondence among their programming elements.

We have developed a preliminary mapping between AspectJ and Hyper/J programming elements. We also have described a set of transformations we have applied and some potential problems and solutions related to the transformation process. A complete and less experimental mapping between AspectJ and Hyper/J would be very useful, as soon as the AspectJ language achieves a stable version and the Hyper/J tool implements everything it promisses.

We believe that, had we started this experiment with hyperslices instead of aspects, the transformation process would pose more difficulties, partially because the composition mechanisms provided by Hyper/J are more powerful and flexible than those provided by AspectJ, partially because we would have some additional work to consider the hyperslices under the base-aspect dichotomy perspective. Clarke [4] proposes Composition Patterns (CPs) as means for achieving separation of concerns at the design level. CPs are clearly inspired in the MDSoC model prescribed by SOP. In [5], she argues that there is a potential for a relatively clean mapping from CPs to Hyper/J code, while the mapping from CPs to AspectJ code may be subject to scattering and tangling in aspects.

We expect that the generic aspect-based design model we have been working on [3, **?**] allows the characterization and comparison of existing and new approaches, as well as an easy mapping from a design model to different programming models that support separation of concerns. Moreover, concern description and concern composition description should be done independently from the integration mechanisms.

References

[1] B. Alwis, S. Gudmundson, G. Smolyn, and G. Kiczales. Coding Issues in AspectJ. In *Int'l Workshop on Advanced Separation of Concerns at OOPSLA*, 2000.

[2] AspectJ Web Page. http://aspectj.org/.

[3] C. Chavez and C. Lucena. Design Support for Aspect-oriented Software Development, 2001. PhD Thesis Proposal, Catholic University of Rio de Janeiro, Informatics Department (in preparation).

[4] S. Clarke. Designing Reusable Patterns of Cross-cutting Behavior with Composition Patterns. In *OOPSLA'00, Work. on Advanced Separation of Concerns*, 2000.

[5] S. Clarke and R. Walker. Mapping Composition Patterns to AspectJ and Hyper/J. In *ICSE 2001, Workshop on Advanced Separation of Concerns in Software Engineering*, 2001.

[6] A. Garcia, M. Cortes, and C. Lucena. A Web Environment for the Development and Maintenance of E-Commerce Portals based on a Groupware Approach, 2001. Information Resources Management Association Int'l Conference (IRMA 2001).

[7] A. Garcia and C. Lucena. An Aspect-Based Object-Oriented Model for Multi-Agent Systems. In *Advanced Separation of Concerns in Software Engineering at ICSE'2001, Toronto*, May 2001.

[8] A. Garcia, C. Lucena, and D. Cowan. Engineering Multi-Agent Object-Oriented Software with Aspect-Oriented Programming, 2001. Submitted to Software: Practice & Experience, Elsevier, April.

[9] A. Garcia, V. Silva, C. Chavez, and C. Lucena. A Comparative Study of Development Techniques for Multi-Agent Object-Oriented Software, 2001. (in preparation).

[10] A. Garcia, V. Torres, C. Lucena, and R. Milidi. An Aspect-Based Approach for Developing Multi-Agent Object-Oriented Systems. In *Brazilian Symposium on Software Engineering, SBES*, October 2001. (to appear).

[11] W. Harrison and H. Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In *Proceedings of OOPSLA'93*, pages 411–428, 1993.

[12] Hyper/J Web Page. http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm.

[13] E. Kendall. Agent Roles and Aspects. In *ECOOP Workshop on Aspect Oriented Programming*, July 1998.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ, 2001. (http://aspectj.org/documentation/overview/aspectj-overview.pdf).

[15] A. Lai, G. C. Murphy, and R. J. Walker. Separating Concerns with Hyper/J: An Experience Report. In *Int'l Workshop on Multi-Dimensional Separation of Concerns in Software Engineering at ICSE*, 2000.

[16] H. Ossher and P. Tarr. Operation-Level Composition: A Case in (Join) Point. In *Intern. Workshop on Aspect-Oriented Programming at ECOOP*, 1998.

[17] M. Skipper. The Watson Subject Compiler & AspectJ (A Critique of Practical Objects). In *Workshop on Multi-Dimensional Separation of Concerns at OOPSLA*, 1999.

[18] P. Tarr and H. Ossher. *Hyper/J User and Installation Manual*. http://www.alphaworks.ibm.com/tech/hyperj.