

A requirements model for quality attributes

Isabel Brito¹, Ana Moreira², João Araújo²

¹Instituto Politécnico de Beja, Beja, Portugal

isabel.sofia@estig.ipbeja.pt

²Departamento de Informática, FCT/UNL, Caparica, Portugal

{amm,ja}@di.fct.unl.pt

Abstract

Quality attributes can be assumptions, constraints or goals of stakeholders. In this paper we present a process to identify and specify quality attributes and to integrate them with functional requirements. The crosscutting nature of some of the quality attributes influences negatively, for example, reusability and traceability in the later stages of the software engineering process. To minimize that influence we start by proposing a template to specify quality attributes at the requirements stage. Then, we extend use cases and sequence diagrams to specify the integration of those attributes with functional requirements.

1 Introduction

Quality attributes, such as response time, accuracy, security, reliability, are properties that affect the system as a whole. Most approaches deal with quality attributes separately from the functional requirements of a system. This means that the integration is difficult to achieve and usually is accomplished only at the later stages of the software development process. Furthermore, current approaches fail in dealing with the crosscutting nature of some of those attributes, i.e. it is difficult to represent clearly how these attributes can affect several requirements simultaneously. Since this integration is not supported from requirements to the implementation, some of the software engineering principles, such as abstraction, localization, modularisation, uniformity and reusability, can be compromised.

What we propose is a model to identify and specify quality attributes that crosscut requirements including their systematic integration into the functional description at an early stage of the software development process, i.e. at requirements.

The rest of this paper is organised as follows. Section 2 presents a model for early quality attributes and discusses its main activities. Section 3 applies our

approach to a case study. Finally, concluding remarks are given in Section 4.

2 A model for early quality attributes

The process model we propose is UML compliant and is composed of three main activities: identification, specification and integration of requirements. The first activity consists of identifying all the requirements of a system and select from those the quality attributes relevant to the application domain and stakeholders. The second activity is divided into two main parts: (1) specifying functional requirements using a use case based approach; (2) describe quality attributes using special templates and identify those that cut across (i.e. crosscutting) functional requirements. The third activity proposes a set of models to represent the integration of crosscutting quality attributes and functional requirements. Figure 1 depicts this model.

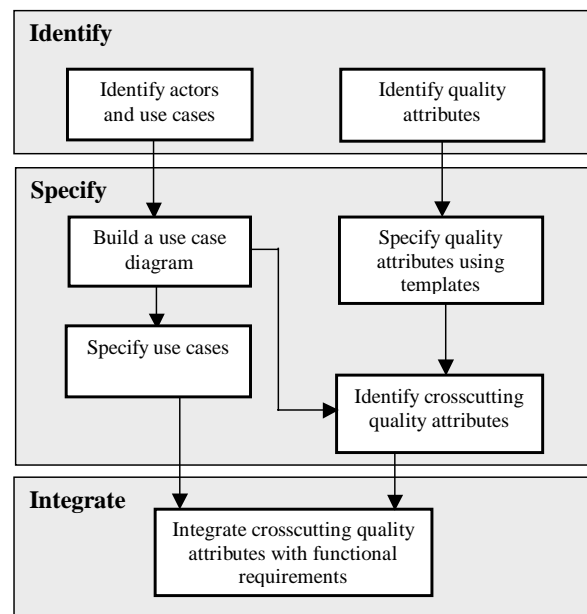


Figure 1: A requirements model for quality attributes

The template we propose to specify a quality attribute was influenced by Mylopoulos *et al.* [8] and Bredmeyer [6, 7] (see Figure 2).

Name	The name of the quality attribute.
Description	Brief description.
Focus	A quality attribute can affect the system (i.e. the end product) or the development process.
Source	Source of information (e.g. stakeholders, documents).
Decomposition	Quality attributes can be decomposed into simpler ones. When all (sub) quality attributes are needed to achieve the quality attribute, we have an AND relationship. If not all the sub quality attributes are necessary to achieve the quality attribute, we have an OR relationship.
Priority	Expresses the importance of the quality attribute for the stakeholders. A priority can be MAX, HIGH, LOW and MIN.
Obligation	Can be optional or mandatory.
Influence	Activities of the software process affected by the quality attribute.
Where	List of models (e.g. sequence diagrams) requiring the quality attribute.
Requirements	Requirements describing the quality attribute.
Contribution	Represents how a quality attribute can be affect by the others quality attributes. This contribution can be positive (+) or negative (-).

Figure 2: Template for quality attributes

To identify the crosscutting nature of some of the quality attributes we need to take into account the information contained in rows Where and Requirements. If a quality attribute cuts across (i.e. is required by) several requirements and models, then it is crosscutting.

The integration is accomplished by “weaving” the quality attributes with the functional requirements in three different ways [1, 3, 5, 12]:

- (1) **Overlap:** the quality attribute adds new behaviour to the functional requirements it transverses. In this case, the quality attribute may be required *before* those requirements, or, it may be required *after* them.
- (2) **Override:** the quality attribute superposes the functional requirements it transverses. In this case, its behaviour substitutes the functional requirements behaviour.

- (3) **Wrap:** the quality attribute “encapsulates” the requirements it transverses. In this case the behaviour of the requirements is wrapped by the behaviour of the quality attribute.

We weave quality attributes with functional requirements by using both standard diagrammatic representations (e.g. use case diagram, interaction diagrams) and by new diagrams.

3 Applying the approach to a case study

The case study we have chosen is a simplified version of the toll collection system implemented in the Portuguese highways [2].

“In a road traffic pricing system, drivers of authorised vehicles are charged at toll gates automatically. The gates are placed at special lanes called green lanes. A driver has to install a device (a gizmo) in his/her vehicle. The registration of authorised vehicles includes the owner’s personal data, bank account number and vehicle details.

A gizmo is read by the toll gate sensors. The information read is stored by the system and used to debit the respective account. When an authorised vehicle passes through a green lane, a green light is turned on, and the amount being debited is displayed. If an unauthorised vehicle passes through it, a yellow light is turned on and a camera takes a photo of the plate.

There are green lanes where the same type of vehicles pay a fixed amount (e.g. at a toll bridge), and ones where the amount depends on the type of the vehicle and the distance travelled (e.g. on a motorway).”

3.1 Identify requirements

Requirements of a system can be classified into functional and non-functional (i.e. quality attributes). Functional requirements are statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. Different types of methods are used to specify functional requirements. Use case driven approaches describe “the ways in which a user uses a system” that is why use case diagram is often used for capturing functional requirements [9]. Quality attributes define global properties of a system. Usually these are only dealt with in the later stages of a software development process, such as design and implementation.

Identify actors and use cases.

For the road pricing system, the actors we identified are:

- **Vehicle owner:** is responsible for registering a vehicle;

- Vehicle driver: comprehends the vehicle, the driver and the gizmo installed on it;
- Bank: represents the entity that holds the vehicle owner's account;
- System clock: represents the internal clock of the system that monthly triggers the calculation of debits.

The following are the use cases required by the actors listed above:

- Register vehicle: is responsible for registering a vehicle and its owner, and communicate with the bank to guarantee a good account;
- Pass single toll: is responsible for dealing with tolls where vehicles pay a fixed amount. It reads the vehicle gizmo and checks on whether it is a good one. If the gizmo is ok the light is turned green, and the amount to be paid is calculated and displayed. If the gizmo is not ok, the light is turned yellow and a photo is taken.
- Enter motorway: checks the gizmo, turns on the light and registers an entrance. If the gizmo is invalid a photo is taken and registered in the system.
- Exit motorway: checks the gizmo and if the vehicle has an entrance, turns on the light accordingly, calculates the amount to be paid (as a function of the distance travelled), displays it and records this passage. If the gizmo is not ok, or if the vehicle did not enter in a green lane, the light is turned yellow and a photo is taken.
- Pay bill: sums up all passages for each vehicle, issues a debit to be sent to the bank and a copy to the vehicle owner.

Identify quality attributes.

Quality attributes can be assumptions, constraints or goals of stakeholders. By analysing the initial of set requirements, the potential quality attributes are identified. For example, if the owner of a vehicle has to indicate, during registration, his/her bank details so that automatic transfers can be performed automatically, then security is an issue that the system needs to address. Another fundamental quality attribute is response time that is a issue when a vehicle passes a toll gate, or when a customer activates his/her own gizmo in an ATM: the toll gate components have to react in time so that the driver can see the light and the amount being displayed. Other concerns are identified in a similar fashion: Multi-user System, Compatibility, Legal Issues, Correctness and Availability.

3.2 Specify functional requirements and quality attributes

The functional requirements are specified using the UML models, such as use cases, sequence and class

diagrams. The quality attributes are described in templates of the form presented in Figure 2.

Build the use case diagram.

The set of all use cases can be represented in a use case diagram, where we can see the existing relationships between use cases and the ones between use cases and actors. Figure 3 shows the use case diagram of the road traffic system.

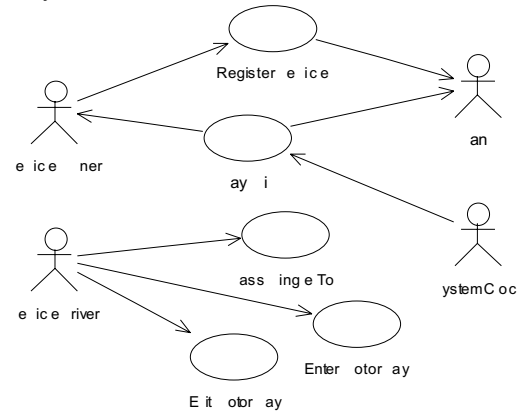


Figure 3. The use case diagram of the Road Traffic Pricing System

Later versions of the use case diagram can show relationships between use cases, in particular some of the use cases share a common set of events in the beginning (which could be shown by adding an extra use case related to the original use cases with the “include” relationship). Extend relationship could also be applied to deal with error situations, for example.

Specify use cases.

Use cases can be described using several techniques, ranging from natural language, scenarios, to formal representations. We prefer to use scenarios, described as a list of numbered steps [10]. A scenario is a particular path of execution through a use case. Use cases can be fully described using a primary scenario and several secondary scenarios, depending on the use case complexity. The primary scenario represents the main path of the use case, i.e. the optimistic view. The secondary scenarios describe alternative paths, including error conditions and exception handling. Each scenario can then be better described using a sequence diagram. A sequence diagram shows the temporal order of interactions between the objects involved in a scenario.

In our case study, we can identify at least two scenarios for each use case. For example, each of the use cases *Pass Single Toll*, *Enter Motorway* and *Exit Motorway*, has a scenario to deal with authorised vehicles and another to deal with non-authorised vehicles. Figure 4 shows the primary scenario “pass single toll gate ok”.

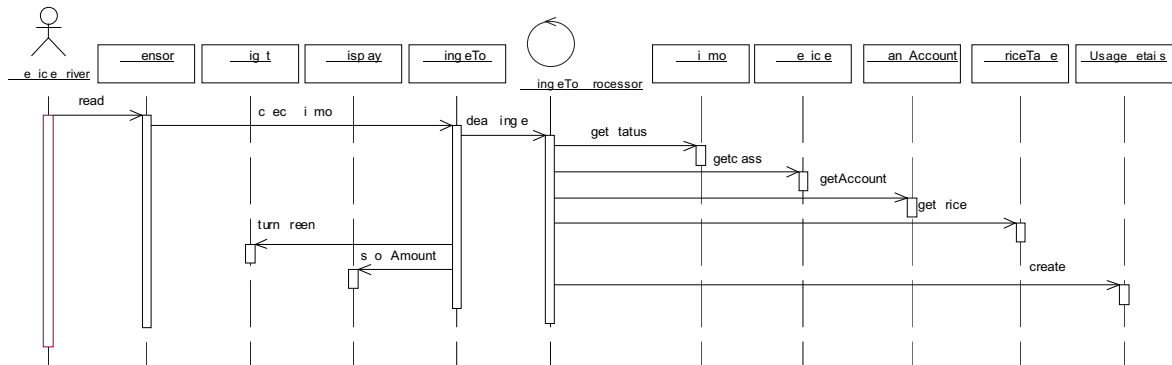


Figure 4. Sequence diagram for primary scenario of *PassSingleToll*

By building a sequence diagram, we can identify the objects in the system needed to handle the corresponding scenario. In our case, we have identified the interface object *SingleToll*, composed of *Display*, *Light* and *Sensor*, the entity objects *Gizmo*, *Vehicle*, *BankAccount*, *PriceTable* and *UsageDetails* and the control object *SingleTollProcessor*. The actors use the interface objects to interact with the system. The entity objects represent the core entities of the system, i.e. the objects with data. Finally, the control objects are the decision makers, i.e. the objects that decide what should be done next.

Specify quality attributes.

To specify quality attributes we use the template presented in Section 2. Let us describe the response-time and security quality attributes (see Figures 5 and 6).

	<ul style="list-style-type: none"> identifier; 1.2 turn on the light (to green or yellow) before the driver leaves the toll gate area; 1.3 display the amount to be paid before the driver leaves the toll gate area; 1.4 photograph the unauthorised vehicle’s plate number from the rear. 2 The bank has to react <i>in-time</i> when customers (re) activates their gizmos through an ATM.
Contribution	(-) to security and to multi-user

Figure 5. Template for Response Time

Name	Response Time
Description	Period of time in which the system has to respond to a service
Focus	System
Source	Stakeholders, original description of the problem
Decomposition	<none>
Priority	MAX
Obligation	Mandatory
Influence	Design, system architecture and implementation
Where	Actors: <i>VehicleDriver</i> , <i>Bank</i> Use cases: <i>RegisterVehicle</i> , <i>PassSingleToll</i> , <i>PassExitToll</i> and <i>PassEntryToll</i>
Requirements	1. A toll gate has to react <i>in-time</i> in order to: <ul style="list-style-type: none"> 1.1 read the gizmo

As can be detected from the templates depicted in Figures 5 and 6, there is a priority conflict between response time and security. A trade-off must be negotiated with the stakeholders to resolve this conflict.

Identify crosscutting quality attributes.

If a quality attribute affects more than one use case (see *Where*), it is crosscutting. This can also be confirmed by analysing the *Requirements* row. For example “response time” cuts across *PassSingleToll*, *PassExitToll* and *PassEntryToll*.

3.3 Integrate functional requirements with crosscutting quality attributes

Integration composes the quality attributes with the functional requirements, to obtain the whole system. We use UML diagrams to show the integration. The

two examples given above (for response time and security) fall into two of the categories already described: overlap and wrapper. We could extend the UML diagrams to represent some quality attributes. For example, the sequence diagram shown in Figure 4 can be extended to show how response time affects a scenario (see Figure 7).

Name	Security
Description	Restricts the access to the system and to the data within the system
Focus	System
Source	Stakeholders
Decomposition	Integrity and Confidentiality. Both have a AND relationship with Security
Priority	MAX
Obligation	Mandatory
Influence	Design, system architecture and

	implementation
Where	Actors: VehicleOwner, Bank Use cases: PayBill, RegisterVehicle
Requirements	The system must: 1. protect the vehicle's owner registration data 2. guarantee integrity in the data transmitted to the bank 3. guarantee integrity on data changed/queried by the operator 4. ...
Contribution	(-) to response time and (+) to multi-user and compatibility.

Figure 6. Template for Security

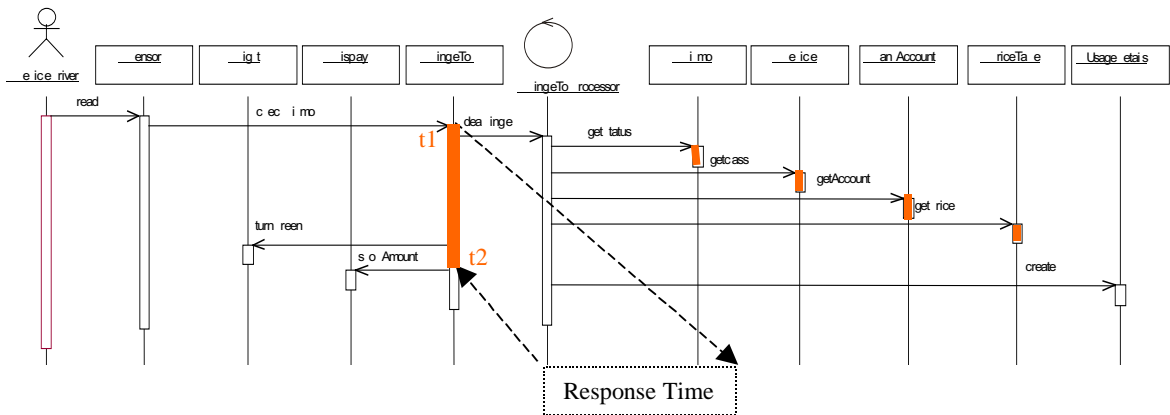


Figure 7. Response Time quality attribute wrap the functional requirements

This figure represents response time wrapping some functional requirements in the sequence diagram. The arrows and the grey rectangles identify the points wherein the constraint applies.

Figure 8 represents security crosscutting use cases. Remember that security is decomposed into confidentiality and integrity. This is an example of an overlap situation, where confidentiality must be used before the execution of the use cases and integrity must be used after it. Dashed lines are used to represent the <<after>> condition and dark lines represent the <<before>> condition.

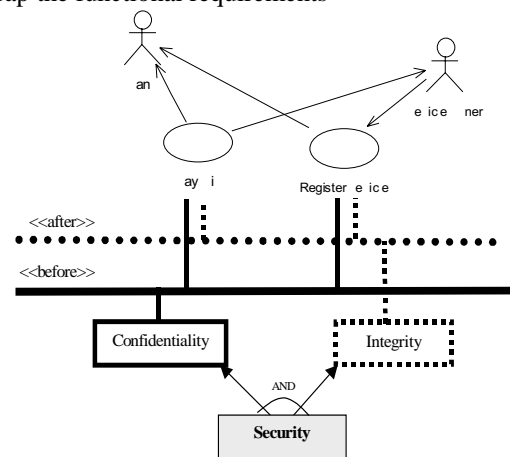


Figure 8. Security overlaps use cases

The (sub) qualities have an AND relationship with Security (see Figure 8).

4. Concluding remarks and future work

This paper presented an approach to identify, specify and integrate requirements. First we identify all the requirements of a system and select from those the quality attributes relevant to the application domain and stakeholders. Afterwards, we specify functional requirements, using a use case based approach, and describe quality attributes using special templates, identifying those that crosscut functional requirements. Finally, we propose a set of models to represent the integration of crosscutting quality attributes with functional requirements.

Last year's major conferences on software engineering and requirements engineering have published some interesting work on quality attributes and crosscutting concerns. Part of our future work is to investigate how other approaches such as ATAM (*Architecture Tradeoff Analysis Method*), composition patterns and goal-oriented requirements engineering relate to our work.

References

[1] Bergmans, L. M. J. and Aksit, M.. "Composing Software from Multiple Concerns: A Model and Composition Anomalies. Multi Dimensional Separation of Concerns in

Software Engineering Workshop, ICSE 2000, Limerick, Ireland, 2000.

[2] Clark, R. and Moreira, A. "Constructing Formal Specifications from Informal Requirements", in *proc. Software Technology and Engineering Practice*, IEEE Computer Society, Los Alamitos, California, 1997, pp. 68-75.

[3] Clarke, S., Walker, R.J. "Composition Patterns: An Approach to Designing Reusable Aspects". In *Proceedings of International Conference On Software Engineering (ICSE 2001)*, Toronto, Canada., 2001.

[4] Constantinides, C. A., Bader, A. and Elrad, T. *An Aspect-oriented Design Framework*. ACM Computing Surveys, March 2000.

[5] IBM Research, MDSOC Software Engineering using Hyperspace <http://www.research.ibm.com/hyperspace/>

[6] Malan, R., and Bredemeyer, D., "Defining Non-Functional Requirements", <http://www.bredemeyer.com/papers.htm>

[7] Malan, R. and Bredemeyer, D., "Functional Requirements and Use Cases", <http://www.bredemeyer.com/papers.htm>

[8] Mylopoulos, J., Chung, L., and Nixon, B., "Representing and Using Non-Functional Requirements: A Process-Oriented Approach", *IEEE Transactions on Software Engineering, Special Issue on Knowledge Representation and Reasoning in Software Development*, Vol. 18(6), 1992, pp. 482-497.

[9] Sommerville, I. and Sawyer, P., "Requirements Engineering, A good practice guide", John Wiley and Sons, 2000.

[10] Schneider G. and Winters J., *Applying Use Cases – A Practical Guide*, Addison-Wesley, 1998.

[11] Suzuki, J. and Yamamoto, Y.. "Extending UML with Aspects: Aspect Support in the Design Phase". AOP Workshop at ECOOP'99, Lisbon, Portugal, 1999.

[12] Xerox PARC, *AspectJ*. <http://www.aspeccj.org>.

From Requirements to Monitors by way of Aspects

Andrew Dingwall-Smith, Anthony Finkelstein
Department of Computer Science
University College London
Gower Street, London, WC1E 6BT UK
{a.dingwall-smith,a.finkelstein}@cs.ucl.ac.uk

Abstract

Using goal driven requirements engineering, requirements are derived from a goal model that captures multiple strategies for satisfying the goals and takes into account environmental constraints on the system. The model is therefore more stable than a conventional requirements document.

We present early work in building a system for runtime monitoring of system goals, as part of normal system operation, so that failure to achieve goals caused by changes in the system environment can be detected and acted on. We make use of Hyper/J to separate instrumentation for monitoring from the core code, and to add instrumentation directly to class files, without the need to modify the core class files. We are currently using a peer to peer networking client as a testbed and we present examples based on this program.

1. Introduction

This position paper presents early work on using requirements specifications to support monitoring of software systems. In the face of a changing environment, a system may no longer be able to meet the goals required of it, due to early assumptions about the environment on which the design has been predicated no longer holding. For this reason, monitoring a system as part of its normal operation is important so that user can be informed of failures or the system configuration changed to take account of environmental changes. This has similarities to [2] which also deals with monitoring a system based on the system goals.

Monitoring systems typically work by instrumenting programs to emit events to the monitoring system, either on the same machine or another machine. Instrumentation of Java programs can be done by inserting instrumentation into the source code or into class files [3]. Instrumentation of class files is done by creating custom tools which manipulate the byte code to insert the instrumentation,

according to high level specifications. This enables the monitoring system to be separated from the program to be monitored.

Our monitoring system makes use of Hyper/J [4],[6] to instrument class files. By using a general purpose tool to support separation of concerns, we eliminate the need to write tools for manipulating bytecode directly.

We are using the Limewire, Gnutella peer to peer networking program as a testbed for our approach. This program is freely available and is written in Java. In Limewire we have a relatively simple system to deal with. However, it is a system which should still be subject to a changing environment as it is affected by the many other Gnutella servants it connects to and issues such as bandwidth and network traffic. This paper uses examples based on our work using Limewire.

2. Goal driven requirements engineering and KAOS

Goal driven requirements engineering is an approach to requirements engineering that aims to capture the rational for requirements and assist in the elicitation of requirements. Goals are elicited from the stakeholders in the system. These goals are used to build a goal model in which the goals are decomposed into sub-goals which describe in greater detail how the goals should be satisfied. Ultimately, requirements can be derived from the goal model. New higher level goals can also be added to capture the purpose of goals.

In general, a set of goals can be decomposed in many ways, providing many possible implementations. The goal model captures multiple goal decompositions and assists in selecting the appropriate decomposition by capturing conflicting goals. The goal model should be more stable than a requirements specification.

The KAOS approach[1], is an example of a goal driven requirements engineering method. This approach specifies goals in terms of an objects model. This provides two views of the system, with one view crosscutting the other.

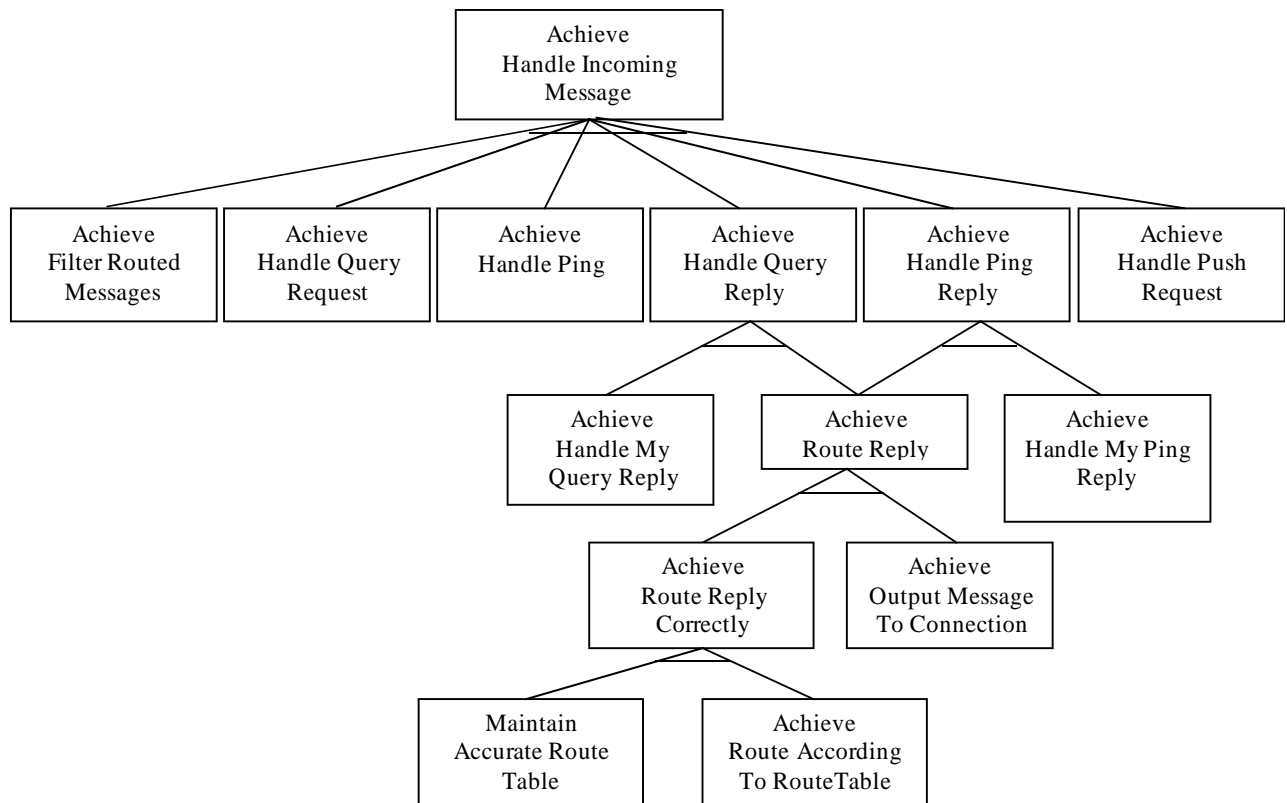


Figure 1. Partial goal decomposition for goal HandleIncomingMessage

KAOS allows goals to be defined using temporal logic if formal specification is desired. The temporal logic formulae refer to objects in the object model. KAOS defines several goal patterns such as Achieve, Maintain and Avoid, which correspond to a certain types of temporal logic formulae. Some common definitions are:

Achieve: $P \Rightarrow \Diamond Q$
 Maintain: $P \Rightarrow \Box Q$
 Avoid: $P \Rightarrow \Box \neg Q$

These logic operators may also have time constraints on them.

Each instance of an achieve goal must eventually be satisfied by the system. The formal specification of the goal may constrain the time in which the goal instance must be achieved.

3. Goal decomposition for Limewire

We have identified goals that stakeholders in the system are likely to require and constructed monitors for

these goals. A partial goal graph for the high level goal 'Handle Incoming Message' is shown in Fig. 1. This goal requires that incoming messages should be handled, in accordance with the Gnutella protocol [5]. This involves sending requests to other connected servants, sending replies back to their originators, responding to requests and so on. Sub-goals are to filter out any unwanted messages and to handle each type of message defined in the Gnutella protocol.

Both the goals 'Handle Query Reply' and 'Handle Ping Reply' have the sub-goal 'Route Reply', as these messages are routed in the same manner. This goal is defined as:

Achieve[RouteReply] Use the route table to route this reply to the connection from which this connection originated.

The goals 'Handle My Ping Reply' and 'Handle My Query Reply' deal with the cases where the reply is a response to a request sent out by our own servant program. The 'Route To Source' goal is further decomposed into the goals 'Routing Reply Correctly' and

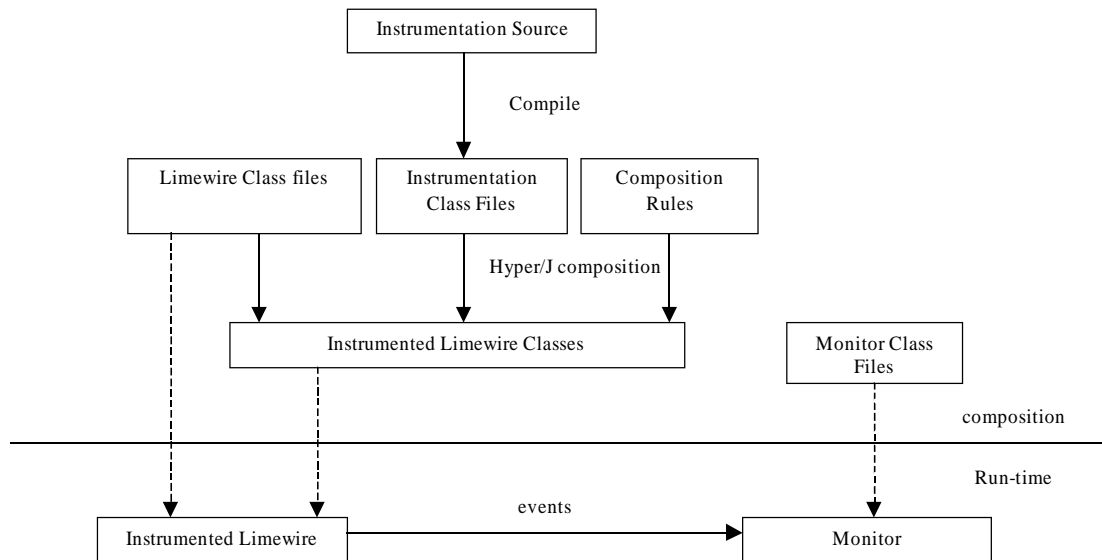


Fig. 2. Monitoring system architecture

'Output Message To Connection'. The first of the goals requires that the messages sent to a connection are sent to the connection held in the route table. The second goal requires that all the messages received are sent to a connection, not dropped. These goals are defined as:

Achieve[RouteReplyCorrectly] Messages should be sent to the correct connection, according to the route table.

Achieve[OutputMessageToConnection] A managed connection which is given a message should output that message to the connection.

The 'Route Reply Correctly' goal can be achieved by sending the reply to the connection in the route table, which is the goal 'Route According To Route Table', and the goal 'Accurate Route Table' which says that the entries in the route table should be correct.

4. Monitoring architecture

The monitoring system architecture is shown in Fig. 2. The inputs to the system are the Limewire class files, instrumentation code, written in Java, the Hyper/J composition rules and the monitor systems class files. The instrumentation source must be compiled by a normal Java compiler as Hyper/J operates on class files. Hyper/J must then be run to integrate the instrumentation classes and the Limewire classes, according to the composition

rules. To run the instrumented program, the Java runtime environment needs access to the original Limewire class files, the instrumented Limewire classes and the monitoring system classes.

The original classes are required as the composition rules tell Hyper/J to only output the classes which need to be instrumented. The instrumented Limewire classes will have the same names as the original Limewire classes so it is important that the class path is set so that the instrumented classes are searched before the original classes.

The monitoring system runs on a single machine using a multi-threaded architecture. Instrumentation is inserted into the program, using Hyper/J, which places events in a queue. At a set interval, a monitoring thread reads the events from the queue and uses the events to determine whether the monitored goal is being satisfied.

5. Hyper/J composition rules

The Hyper/J composition rules, Fig. 3, introduce two dimensions of concern, in addition to the existing Object dimension, which is created automatically. In the 'Goal' dimension, each monitor belongs to the concern corresponding to the goal it is trying to monitor. In the 'Aspect' dimension, the original Limewire classes are in the 'Core' concern. Only those classes which need to be instrumented are imported into the hyperspace, using 'as in package' at lines 14 and 15 of the specification file.

In Java, a class which has no constructor defined for it

```

1. -concerns
2.   package monitor.outputmessage : Goal.OutputMessage
3.   operation monitor.outputmessage.ManagedConnection.<init> : Goal.None
4.   operation monitor.outputmessage.PingReply.<init> : Goal.None
5.   operation monitor.outputmessage.QueryReply.<init> : Goal.None
6.   operation monitor.outputmessage.RouterService.<init> : Goal.None
7.
8.   package monitor.routecorrectly : Goal.RouteCorrectly
9.   operation monitor.routecorrectly.ManagedConnection.<init> : Goal.None
10.  operation monitor.routecorrectly.MessageRouter.<init> : Goal.None
11.  operation monitor.routecorrectly.RouterService.<init> : Goal.None
12.  operation monitor.routecorrectly.RouteTable.<init> : Goal.None
13.
14.  package com.limegroup.gnutella as in package monitor.outputmessage : Aspect.Core
15.  package com.limegroup.gnutella as in package monitor.routecorrectly : Aspect.Core
16.
17. -hypermodules
18.  hypermodule MonitoredLimewire
19.    hyperslices:
20.      Aspect.Core,
21.      Goal.OutputMessage,
22.      Goal.RouteCorrectly;
23.  relationships:
24.    mergeByName;
25.
26.    order action Goal.RouteCorrectly.ManagedConnection.handlePingReply
27.    before action Aspect.Core.ManagedConnection.handlePingReply;
28.
29.    order action Goal.RouteCorrectly.ManagedConnection.handleQueryReply
30.    before action Aspect.Core.ManagedConnection.handleQueryReply;
31.
32.    order action Goal.OutputMessage.ManagedConnection.handlePingReply
33.    before action Aspect.Core.ManagedConnection.handlePingReply;
34.
35.    order action Goal.OutputMessage.ManagedConnection.handleQueryReply
36.    before action Aspect.Core.ManagedConnection.handleQueryReply;
37.
38.    order action Goal.OutputMessage.PingReply.writePayload
39.    after action Aspect.Core.PingReply.writePayload;
40.
41.    order action Goal.OutputMessage.QueryReply.writePayload
42.    after action Aspect.Core.QueryReply.writePayload;
43.
44.    order action Goal.RouteCorrectly.MessageRouter.handlePingReply
45.    before action Aspect.Core.MessageRouter.handlePingReply;
46.
47.    order action Goal.RouteCorrectly.MessageRouter.handleQueryReply
48.    before action Aspect.Core.MessageRouter.handleQueryReply;
49.
50.  end hypermodule;

```

Fig. 3. Hyper/J composition rules

has a default constructor generated by the compiler. This can cause problems with Hyper/J, as classes without constructors are often not intended as stand alone classes. The default constructors generated for the goal monitor classes need to be explicitly assigned to the Goal.None concern so that they are excluded from the composition.

The relationships section specifies the merge by name composition strategy, meaning that classes with the same name are merged together by merging fields and methods with the same name. The order relationships specify the relative order of method bodies in the merged methods, in

the cases where the order is important.

The goals which are being monitored can be changed by simply adding or removing those concerns to the composition and then running Hyper/J again.

The classes into which the instrumentation has to be composed are obtained from the goal specifications. The goals are defined in terms of a domain-level object model. The domain-level objects then have to be related to the implementation object model. At present, we only do this informally, using the natural language definition of a goal and the objects that are referred to in this definition. Using

the temporal logic specification used in KAOS, and a formal object model, may allow us to formally derive this relationship. As an example, the goal 'Output Message To Connection' refers to the objects 'Managed Connection' and 'Message'. In the Hyper/J composition rules, instrumentation for this goal is composed into the Limewire classes 'Managed Connection', 'Ping Reply' and 'Query Reply'. 'Ping Reply' and 'Query Reply' are subclasses of the Limewire 'Message' class. This shows the correspondence between the Limewire classes which are instrumented and the goal definition.

6. Monitor design

We have implemented monitors for the goals 'Output Reply To Connection' and 'Route Message According To Route Table'. Since both of these goals fit into the achieve pattern, the monitors are quite similar. In the case of the 'Output Message To Connection' goal, whenever the 'Managed Connection' class receives a reply message, the monitor stores the record of that event. The record includes the time of the event and a reference to the Message object that was received. At set intervals, the monitor checks every stored record to see if the time since it occurred is greater than the time constraint on the goal. If it is then the record is removed from the store and the monitor reports that the program has failed to achieve the goal.

Whenever a message is sent to the connection by the 'Query Reply' or 'Ping Reply' classes, the monitor searches the record of events for one with a matching Message reference. If one is found then it indicates that the goal has been successfully achieved. Otherwise, the time constraint on the message must already been violated and the failure reported, so no further action is necessary.

7. Conclusions and further work

Our system currently only monitors goals which match the KAOS achieve pattern. Implementing monitors for other patterns such as maintain and avoid should be fairly trivial.

We also intend to look at monitoring non-functional

goals and soft goals. Non-functional goals seem particularly suited to our approach. Soft-goals, that is goals that do not have defined conditions for achievement, are more difficult. For soft goals, it is not obvious what sort of information is useful to the user or the program itself.

Our goal is to generate monitors from high level specifications, that is, generating monitors from the temporal logic used in KAOS. To date we have used the specification as a guide but hard coded the monitors to give us a sense of whether the approach might work.

8. Acknowledgements

This research has been supported by BTexact and EPSRC in their collaborative programme 'Generative Software Development'. We are grateful for their generous support.

9. References

- [1] A. Dardenne, A. van Lamsweerde, and S. Fickas. "Goal-directed requirements acquisition", *Science of Computer Programming*, 20:3-50, 1993
- [2] M. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard. "Reconciling System Requirements and Runtime Behaviour", *Proc. IWSSD'98 - 9th International Workshop on Software Specification and Design*, IEEE CS Press, April 1998.
- [3] M. Kim, S. Kannan, I. Lee, O. Sokolsky and Mahesh Viswanathan. "Java-MaC: a Run-time Assurance Tool for Java Programs", *Proc. RV'01- 1st Workshop on Runtime Verification*, *Electronic Notes in Computer Science*, 55(2), 2001.
- [4] P Tarr, H Ossher, W Harrison and SM Sutton, Jr. "N degrees of Separation: Multidimensional separation of concerns", *Proc. ICSE 99*, IEEE, May 1999, ACM press
- [5] "The Gnutella Protocol Specification v0.4", http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf
- [6] "Hyper/J" <http://www.alphaworks.ibm.com/tech/hyperj>

Mining Aspects

Neil Loughran, Awais Rashid

Computing Department, Lancaster University, Lancaster LA1 4YR, UK
{loughran | awais} @comp.lancs.ac.uk

Abstract. The mining of existing assets is an important concern for software developers and organisations. The availability of tools which allow fine grained queries to be performed on these assets will improve productivity by allowing developers to locate and adapt assets efficiently. Mining of assets can take part at many different stages throughout the software development lifecycle. Typical assets for mining could include program code, designs, system architectures, specifications, algorithms and the like. Aspects are particularly suitable for mining as they allow system wide concerns such as synchronisation, logging, debugging and the like to be encapsulated into a single construct, making them ideal candidates for reuse. This paper discusses the increasing demand for tools that support the mining of existing assets with a special focus on Aspect-Oriented Software Development (AOSD).

1 Introduction

This paper discusses the increasing demand for tools that support the mining of existing assets with a special focus on Aspect-Oriented Software Development (AOSD).

As software systems increase in size and become more complex, the need for quality software to be produced within cost and schedule constraints has become a prime concern for software developers and organisations. The reuse of existing assets has been demonstrated to greatly reduce software development costs, as well as improve productivity and quality [1]. Being able to store assets has other benefits such as being able to store domain knowledge, reuse context, test cases and results, and other verification and validation data.

The term ‘mining existing assets’ generally refers to the locating of useful information from an organisation’s asset base for reuse in new applications. Typical candidates for mining include program code, designs, system architectures, specifications, algorithms and the like. Repositories for the storage of these assets can play a central role in the development of new systems. However, whilst having a solid core base of these assets is important, navigation, storage and retrieval in a meaningful manner is not a simple task due to the underlying data representations involved. If it takes too long to locate an asset then the benefits of reuse are not being fully utilised. Various approaches (in increasing complexity) in use range from the storage of keywords (meta-data) along with the asset to the mapping of an asset to a suitable representation within the asset repository (this usually requires mapping the asset to its representation in the underlying database model).

In this paper, we focus on the mining of one particular category of assets: the aspects. This is of particular interest due to the following reasons:

Aspects seek to solve some general architectural problems which are common from system to system in a modularised way. Hence, effective storage and mining support can provide a higher degree of reuse.

The crosscutting nature of aspects dictates that an effective mining approach is not only able to find suitable aspects but also assist in determining the context of their original use and influence within the new context.

Effective asset mining can help trace an aspect and, therefore, its variations (to suit the new application context) and their impact throughout the software life cycle. While this is a feature desirable for all assets, the crosscutting nature of aspects makes such traceability critical for effective reuse and change impact determination.

There is an increasing interest from the software engineering community in this new paradigm. Therefore, it is only natural that support for asset mining be extended to this new paradigm.

The next section further motivates the need for aspect mining. Section 3 discusses the nature of tool support required and some initial prototypes for the purpose. Note that the prototypes, at present, only support mining of code level aspects due to the less well-defined nature of aspects at earlier development stages. However, the discussion in this paper is based on the view that aspect mining applies to aspects at all development stages. Also, as discussed in section 3.4, the approach used in the prototypes also forms a suitable candidate for mining early aspects.

2 Need for Aspect Mining

AOSD is a new software development paradigm which employs special abstractions, the aspects, to modularise concerns that cut across other parts of a system. Examples of such crosscutting concerns include synchronisation, debugging, logging, memory management, security and persistence. The modularisation of such concerns with AOSD techniques renders them ideal candidates for reuse. By having aspects available which have specified tasks and attributes that can be used system wide we can reuse them with little or no alteration in other systems. However, for such variation and reuse to be effective it is essential that not only suitable aspects are retrieved but also support is available to help determine their original context of use and their influence within the new application context. This is critical as with mechanisms such as pattern matching employed in techniques such as AspectJ [2] it is possible that reuse will result in matching modules to which the aspect's behaviour should not apply or ignoring modules to which it should. In case of variation of requirements between the two application contexts it is essential that this variation is traced from early on in the life cycle through to the implementation and evolution stages.

It has been said that design and testing account for a major portion (in some cases as high as 80%) of the system costs [3]. Effective aspect mining can provide more effective reuse of existing assets. Besides, a significant number of tests used in the verification of these assets can be reused. As mentioned earlier the crosscutting nature of aspects makes such verification and validation an expensive activity. Aspect storage and mining can help reduce these costs.

As pointed out in [4] several aspects follow easily recognisable patterns and so can be adapted relatively easily to new contexts. An example of this would be the observer pattern or a simple tracing aspect which logs the instantiations and calls exhibited by a system during testing. The existence of such reusable (and adaptable) patterns further strengthens the argument for aspect storage and mining.

The possibility of mining the asset base for existing aspects and adapting them to new or slightly different application contexts at lower effort and cost brings exciting possibilities to support variation within software product lines. Software product lines are concerned with the creation of families of products, with each product sharing a common set of features. For instance a common asset in a software product line could be a networking module that all the products in the family share. The variation point in this instance might be the network protocols which that particular module is capable of. The aspect could simply weave in the various capabilities required for that particular software product.

3 Tool Support

A basic premise for any mining tool is that it must be able to locate the desired assets efficiently. If the underlying data structures are too complex and the database is populated with a very large number of assets then the mining operation could take too long to perform. Similarly, if the underlying data representation is too simple then the potential benefit of fine-grained querying is minimal resulting in too many records (most of them undesirable) being retrieved.

There are a number of approaches to consider for the development of a framework to support the mining of aspects at different stages of the software development lifecycle. We consider three of these approaches in some detail. These include:

- *Direct* aspect storage coupled with meta-data
- Mapping of aspect anatomy to the database model
- Hybrid approach

3.1 *Direct* Aspect Storage Coupled with Meta-data

This approach entails storing the aspect as a binary/character object along with some useful meta-data that describes the object in a coarse grained fashion. The user is then able to query this meta-data in order to retrieve the aspects. Fig. 1 outlines a possible implementation on a requirements aspect.

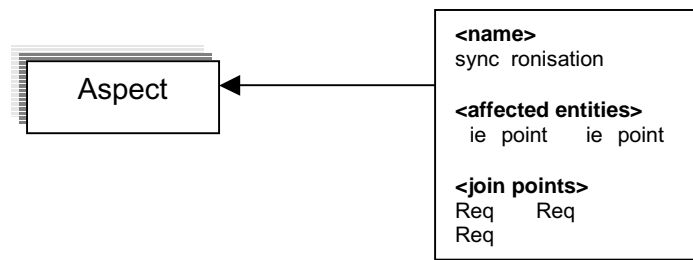


Fig. 1. Requirements aspect with associated meta-data representation

Advantages:

- easy to implement;

- fast and efficient.

Disadvantages:

- loss of aspect representation in the database;
- needs effective capturing of meta-data and its evolution as the aspect changes in line with different application contexts;
- fine-grained variation and traceability is severely constrained by loss of aspect representation;
- query complexity constrained by meta-data;
- aspect constrained for use in the particular AOSD approach used to develop it.

3.2 Mapping of Aspect Anatomy to the Database Model

This approach involves fine-grained mapping of an aspect to the underlying database model e.g. an object model or a relational model. The user is then able to query properties of an aspect in a flexible fashion and, hence, perform complex fine-grained queries. Simple examples of such queries include finding join points influenced by an aspect or tracing a fine-grained change in a specification aspect to its corresponding design and implementation aspects (at a fine-granularity). An aspect mapped to the relational database model (simplified for the purpose of this paper) is displayed in fig 2.

A side benefit of this approach is that the aspect storage structure might relate to one particular AOSD technique but upon retrieval the aspect may be transformed for use in a different AOSD mechanism. [5, 6], for example, propose algorithms to map aspects in AspectJ to Hyper/J [7] and vice versa. Similarly, queries based on a different AOSD technique can be transformed to comply with the aspect representations in the database. As AOSD techniques at earlier development stages mature, we envisage that such algorithms will be developed for mapping between them and exploited during retrieval of aspects at these stages.

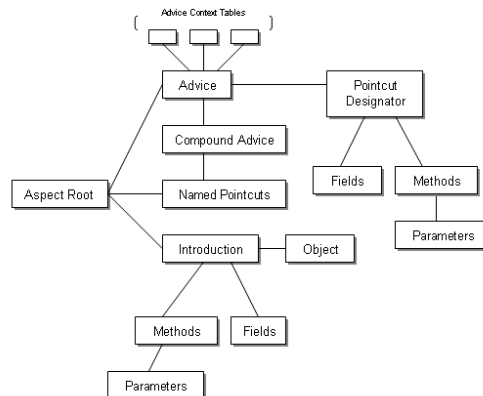


Fig. 2. Simplified model of aspect mapped to relational schema

Advantages:

- aspect representation is retained in the database;
- fine-grained querying possible;

- resilient to changes in the aspect as there is no associated meta-data;
- can be relatively easy to adapt to other AOSD techniques.

Disadvantages:

- complex to implement;
- difficulty in mapping certain aspectual constructs such as combinational logic;
- potentially resource intensive.

3.3 Hybrid Approach

Perhaps the best option would be a hybrid of the previously mentioned approaches. The hybrid solution could involve only storing the most important parts of an aspect's anatomy into a database together with the complete aspect itself as a binary or character object.

Advantages:

- a reasonable degree of fine-grained querying possible;
- some aspect representation retained in database;
- relatively easy to implement;
- good efficiency.

Disadvantages:

- some restrictions on the types of querying possible;
- aspects constrained to a particular AOSD technique.

3.4 Existing prototypes

To date we have developed two prototypes to store, retrieve and mine AspectJ aspects. One of these prototypes, PersAJ, is based on an object database [8] while the other employs a relational database. Our choice of aspects has been constrained to the implementation level due to the lack of well-defined semantics of aspects at earlier development stages.

We have chosen to employ the approach based on mapping the aspect anatomy to the database model as it provides greater flexibility in terms of aspect mining (fine-grained queries are possible) and evolution of the persistent aspect structure and retrieval mechanisms in line with the continuously evolving nature of AOSD techniques [9]. During our experiments with code level aspects we have found fine-grained mapping and querying of aspects to be useful in determining their original context and their influence and effectiveness in the application context in which they are to be reused. We have also found that the fine-grained approach makes variations an easier task. We are of the view that while aspects at earlier stages mature it will be best to use a similar mechanism for storage and mining of these early aspects. Once, the semantics and representation of aspects at the various development stages have stabilised it will be more efficient to move to the hybrid approach.

4 Conclusion

It has been demonstrated with this paper that the mining of aspects will become an important concern where reuse is required. Being able to recover the desired aspects accurately and efficiently for reuse or in order to create new ones will have a beneficial effect on the software community in terms of time to market, and it remains to be seen what other benefits and solutions AOSD will bring. Certainly AOSD's ability to localise crosscutting concerns to single constructs can benefit the software community by effectively making such concerns easier to read and alter in response to changing requirements.

Having the ability to run very fine-grained queries may sound good in practice but it would have to be up to the end user to judge if the benefits of that approach outweigh its demerits. It has been said that the reuse of small grained software assets such as subroutines or small programs is rarely economical with the most useful assets for mining being those which make up large patterns of interoperation throughout architecture. However, the latter could certainly describe the functionality of any given aspect construct.

The table in fig 3 summarises the attributes which the different data modelling approaches possess.

Approach	Direct Storage with Meta-data	Relational Database Mapping	Hybrid
Complexity of Implementation	impe	Comp e	impe to oderate
Query Flexibility	o	ery ig	oderate
Performance	ig	ependent on uery Comp e ity	oderate to ig
Typical Aspect Candidates	Requirements Aspects esign ocumentation Arc itectura ode s	Aspectua Code atterns	Requirements Aspects esign ocumentation Arc itectura ode s Aspectua Code atterns

Fig. 3. Comparison of existing approaches

Perhaps the best solution and compromise would be the use of a hybrid data structure where efficiency and some ability for fine-grained queries co-exist and the structures are not too complex to design and implement. The decision as to which data modelling technique best serves the aspect mining needs of the user ultimately depends upon the resources, context and reuse policies in place within the organisation.

References:

1. Cusumano, M. *The Software Factory: A Historical Interpretation*, IEEE Software (March 1989) pp. 23-30:
2. Xerox PARC, USA, *AspectJ Home Page*, <http://aspectj.org/>
3. Cusumano, M., *Japans Software Factories*, Oxford University Press, 1991:
4. Clarke, S. and Walker, R.J. *Composition Patterns: An Approach to Designing Reusable Aspects*. *International Conference on Software Engineering (ICSE)*, 2001:
5. Chavez, C., Garcia, A.F., and Lucena, C.J.P. *Some Insights on the Use of AspectJ and Hyper/J*. *Workshop on Aspect-Oriented Programming and Separation of Concerns, Lancaster University, UK*, 2001: Cooperative Systems Engineering Group, Technical Report:
6. Clarke, S. and Walker, R.J. *Mapping Composition Patterns to AspectJ and Hyper/J*. *ICSE 2001 Workshop on Advanced Separation of Concerns in Software Engineering*, 2001:
7. IBM Research, *Hyperspaces*, <http://www.research.ibm.com/hyperspace/>
8. Rashid, A. *On to Aspect Persistence*. *2nd International Symposium on Generative and Component-based Software Engineering (GCSE)*, 2000: Springer-Verlag, Lecture Notes in Computer Science 2177: 26-36;
9. Rashid, A. *Weaving Aspects in a Persistent Environment*, ACM SIGPLAN Notices, Feb. 2002

Architectural Aspects

Position paper submitted to the AOSD2002 Workshop on
Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design

Kim Mens

Département INGI, Université catholique de Louvain
Place Sainte-Barbe 2, B-1348 Louvain-la-Neuve, Belgium
E-mail: Kim.Mens@info.ucl.ac.be

February 27, 2002

Abstract

We make a case for the relevance of the ideas of aspect-oriented programming at the architectural level. Traditional approaches to software architecture often assume that a software system exhibits a single architecture, of which the elements map more or less directly to implementation-level components. We claim that multiple architectural views, that may crosscut the implementation structure, can provide a better insight in the overall structure, organization and functionality of a software system than one single architecture which is often strongly biased towards the implementation structure of the system. The elements in such a crosscutting architectural view can be regarded as a kind of *architectural aspects* that describe how the element crosscuts the implementation structure.

Introduction

When designing a building, architects do not make one single plan that describes the overall structure of the entire building. Instead, they use many different plans that each focus on a single aspect or view of the building: front and side views, floor plans, cross-sections, foundation, drainage system, electrical wiring, central heating, and so on. Not

only do these plans address different concerns, they are also supposed to be used by different persons: clients, bricklayers, electricians, plumbers, and so on. Many of these plans are clearly crosscutting. For example, a client's request to add an extra window (based on a side view of the building) may require parts of the electrical wiring to be reconfigured, since the wiring is often incorporated in the walls. It may even require a partial restructuring of the building, because a window is not a load-bearing structure. It is the architect's job to try and construct a building that optimally satisfies the different constraints imposed and concerns addressed by all these plans.

In contrast with this accepted approach in building architecture, many approaches in the domain of software architecture (e.g., [11, 13]) still seem to assume that software architectures have a direct mapping of their architectural elements to source-code, design-level or physical artifacts and their dependencies. We refer to such architectures as *application architectures* because they focus on the actual implementation structure of a software application. Application architectures describe what the implementation components are and how they are interrelated.

Although such application architectures provide good insights into the structure of a software system and thus facilitate detailed

design and implementation as well as evolution and maintenance of the system, there is, in general, no reason why a software architecture *should* resemble the application structure. The building blocks of a software architecture are merely abstract concepts that are meaningful for the application domain. An architecture is a relation (or structure) over such concepts. Therefore, apart from the application architecture, many other kinds of architectures are imaginable and needed. For example, an architecture focusing on specific aspects of the system such as user interaction, distribution and error handling, or even architectures addressing domain-specific concerns such as rule-based interpretation (in the domain of rule-based systems). Such architectures, however, often *crosscut* the application structure or application architecture. Furthermore, even the application structure itself can be described from different viewpoints, for example, from a data-flow or from a control-flow perspective.

The idea of having not only an application architecture but also many other overlapping and crosscutting architectures that address specific concerns is obviously inspired by the research on *aspect-oriented programming* (AOP) [5]. AOP tries to solve the problem that when a software system is structured according to its base functionality, adding aspects which crosscut this structure often requires system-wide changes. This problem is caused by what Tarr et. al. [14] call the *tyranny of the dominant decomposition*: typically, a software system is decomposed according to one ‘dominant’ concern and other concerns that crosscut this basic functionality are difficult to incorporate in the software. In AOP, there is no dominant concern. The base program and several aspect programs are all implemented separately and are then merged into one single executable program. In the same spirit, Ossher and Tarr suggest to adopt a software development approach which allows a simultaneous decomposition according to multiple, potentially over-

lapping concerns or dimensions. Approaches such as AOP, subject-oriented programming [3], adaptive programming [7] and composition filters [1] can, in some sense, be regarded as a special case of their approach.

With this position paper we want to illustrate the relevance and importance of the above ideas at the architectural level. In fact, we make two different claims:

1. A software system does not necessarily have one single (dominant) architecture, but should be described by several (potentially overlapping) architectural views, each providing their own perspective on the software system.
2. The elements in an architectural view do not need to map directly to implementation or design-level components but may actually crosscut the software. As such, they can be considered as a kind of ‘architectural aspects’.

Terminology

Many authors [2, 4, 6, 11, 12] consider a software architecture merely as a structural description of the interaction among the software components of which the system is constructed. In this view, there is no objection against using the term *component* at the architectural level. However, because of our position that architectural views do not necessarily require a direct mapping of the architectural elements to design-level, implementation-level or physical components, we are *not* tempted to adopt this terminology. Not only is the term (software) *component* already heavily overloaded, most definitions of components seem to agree at least on the fact that a software component is some localized, reusable and replaceable piece of implementation of a software system. Extending the usage of the term, to denote architectural elements that crosscut the design or implementation, would only give rise to confusion. In-

stead, we prefer to use the term *architectural aspect* to denote such architectural elements. So instead of talking about architectural components and connectors (as, for example, in [12]), we will talk about architectural *aspects* and *relations* respectively.

Experiments

We validate our claims by means of some experiments that have been conducted in the context of our Ph.D. research. We try to declare the architecture of some software system from different points of view, and automatically check conformance of the implementation of that system to these architectural views. The system we considered was SOUL, a rule-based programming environment, implemented entirely in Smalltalk.

Multiple architectural views

To validate our claim that a software system may have multiple, potentially overlapping, architectural views we show two complementary views for the SOUL system: the ‘user interaction’ architectural view and the ‘rule interpreter’ architectural view. Both views are valid descriptions of the system, in the sense that conformance of the system’s source code to these descriptions was verified. Due to space limitations, however, we will not discuss the details of how conformance checking was achieved.

The ‘user interaction’ architectural view, depicted in Figure 1, focuses mainly on the interaction of a user with the SOUL system. We summarize only some of its most important aspects here. The SOUL environment comes with a predefined set of *User Applications* that are activated when certain events are triggered by the user in some *Input Window*. *Auxiliary Applications* are applications that are created by *User Applications* or other *Auxiliary Applications* to do part of their computation. After computation, a *User Application* typically produces a *Query Result*,

which is not returned to the user directly, but presented in an *Output Viewer* for easy browsing and inspecting of the result.

Since SOUL is a rule-based environment, a second important architectural view is the rule interpreter architecture, depicted in Figure 2. Due to space limitations, for details on this architecture we refer to [12].

It is important to mention, though, that both architectural views are partially overlapping. For example, they both contain the concepts *Rule Interpreter* and *Knowledge Base*.

Crosscutting architectural views

To support the claim that the concepts in an architectural view do not necessarily map directly to implementation artifacts, but may actually crosscut the entire software implementation, we revisit the rule interpreter architectural view of the previous subsection.

When trying to map the elements of the rule interpreter architectural view to the actual SOUL implementation, we noticed that the elements in this architectural view did not always map straightforwardly to the classes or other artifacts in the implementation. For some views, a *crosscutting* mapping from their architectural elements (or ‘architectural aspects’) and relations to their corresponding implementation artifacts and relationships was needed.

Consider as an example the *Rule Interpreter* architectural aspect. Intuitively, this aspect corresponds to all implementation artifacts that address the concern “interpretation of queries” in the implementation of the SOUL rule-based environment. Unfortunately, these artifacts were not localized in the implementation, but were spread throughout the entire implementation. In fact, the implementation was decomposed according to the syntax of SOUL’s logic language. Every node in the abstract grammar was represented by a different class, each containing one or more methods implementing part of the interpretation process. Thus, the *Rule Interpreter*

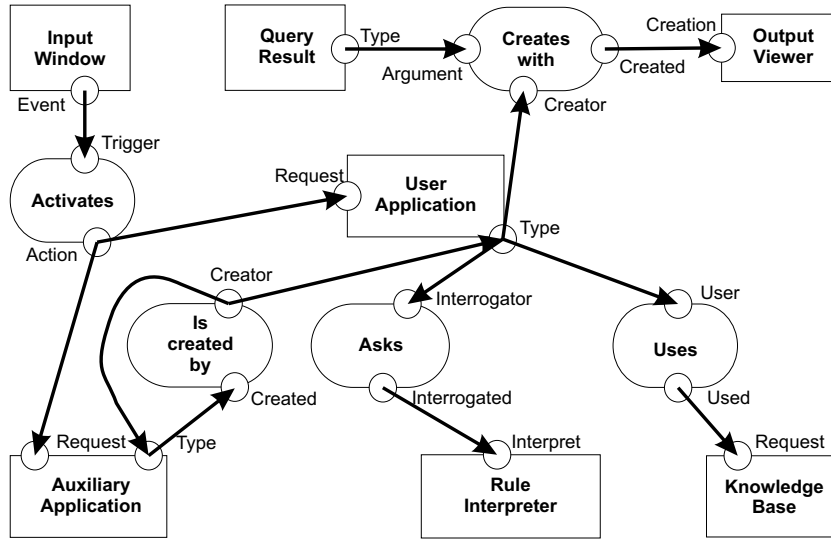


Figure 1: User interaction architectural view

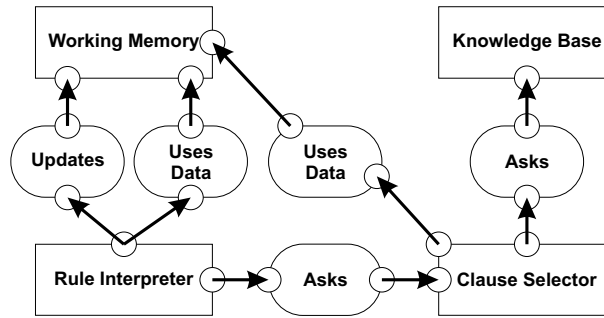


Figure 2: Rule interpreter architectural view

aspect crosscuts the implementation as it is mapped to all these methods belonging to many different classes.

Summary

Experiments conducted in the context of our research on architectural conformance checking made us realize that an architecture which provides a high-level view of some aspect of the design of a software system, does not necessarily need to map directly to the source code, but may crosscut it. Furthermore, many of these crosscutting architectural views may be needed to provide a better picture of the overall structure, organization and functionality of a software system. These architectural

views may even be partially overlapping. In short, we think that the architectural research community could benefit from adopting some of the ideas of the aspect-oriented programming community.

Future work

The architectural elements — or architectural aspects — in a crosscutting architectural view can be regarded as a kind of ‘pointcuts’ in the sense that they describe how the architectural element crosscuts the implementation structure. We are working on a formalism to model these ‘pointcuts’ as intentionally declared views on the source code.[10, 8, 9] The formalism is supposed to facilitate con-

formance checking of an implementation to an architecture that is build up from such crosscutting architectural elements. While working on this formalism, we have also experienced that these source-code views are very useful as a starting point for doing code generation.[15] As such we obtain a kind of ‘architecture-driven’ aspect-oriented programming. The architectural aspects define the pointcuts where additional code should be weaved. One can see this as an ordinary kind of AOP where the aspects happen to be architectural elements. But in fact, an explicit architecture description offers something extra to AOP too as it defines an explicit interaction structure among the different aspects. As such, this approach might prove beneficial for both the software architecture and the AOP community.

References

- [1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object-based Distributed Processing*, volume 791 of *Lecture Notes in Computer Science*, pages 152–184. Springer-Verlag, 1993.
- [2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison Wesley Longman, 1998.
- [3] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications, OOPSLA 1993*, pages 411–428. ACM Press, 1993.
- [4] P. Inverardi, A. L. Wolf, and D. Yankelevich. Checking assumptions in component dynamics at the architectural level. In *Coordination Languages and Models*, volume 1282 of *Lecture Notes in Computer Science*, pages 46–63. Springer-Verlag, September 1997. Second International Conference, COORDINATION 1997, Berlin, Germany.
- [5] G. Kiczales. Aspect-oriented programming. In *European Conference on Object-Oriented Programming, ECOOP 1997*. Springer, 1997. Invited presentation.
- [6] J. Kramer and J. Magee. Exposing the skeleton in the coordination closet. In *Coordination Languages and Models*, volume 1282 of *Lecture Notes in Computer Science*, pages 18–31. Springer-Verlag, September 1997. Second International Conference, COORDINATION 1997, Berlin, Germany.
- [7] K. J. Lieberherr. *Adaptive Object-Oriented Software. The Demeter Method with propagation patterns*. PWS Publishing Company, 1996.
- [8] K. Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, October 2000.
- [9] K. Mens and T. Mens. Codifying high-level software abstractions as virtual classifications. Workshop paper vub-prog-tr-00-14, Programming Technology Lab, Vrije Universiteit Brussel, Belgium, March 2000. ECOOP 2000 Workshop on Objects and Classification: a Natural Convergence.
- [10] K. Mens, R. Wuyts, and T. D’Hondt. Declaratively codifying software architectures using virtual software classifications. In *Proceedings of TOOLS Europe 1999*, pages 33–45. IEEE Computer Society Press, 1999. TOOLS 29 — Technology of Object-Oriented Languages and Systems, Nancy, France, June 7-10.

- [11] M.-C. Pellegrini. Dynamic reconfiguration of CORBA-based applications. In *Proceedings of TOOLS Europe 1999*, pages 329–340. IEEE Computer Society Press, 1999. TOOLS 29 — Technology of Object-Oriented Languages and Systems, Nancy, France, June 7-10.
- [12] M. Shaw and D. Garlan. *Software Architecture — Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
- [13] P. Stevens and R. Pooley. *Using UML — Software Engineering with Objects and Components*. Addison Wesley, 1999. Updated edition.
- [14] P. Tarr, H. Ossher, W. Harrison, and J. S. M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering (ICSE 1999)*, 1999.
- [15] T. Tourwé and K. De Volder. Using software classifications to drive code generation. Workshop paper, Programming Technology Lab, Vrije Universiteit Brussel, Belgium, March 2000. ECOOP 2000 Workshop on Objects and Classification: a Natural Convergence.

Towards the Identification of Concerns in Personalization Mechanisms Via Scenarios

Cláudia Mesquita, Simone Diniz Junqueira Barbosa, Carlos José Pereira de Lucena
Informatics Department, PUC-Rio
R. Marquês de São Vicente, 225
Gávea – Rio de Janeiro – RJ 22453-900
Brazil
e-mail: {mesquita, sim, lucena}@inf.puc-rio.br
+55 21 3114-1500 ext. 4430

Abstract

In this position paper we discuss the identification of concerns related to personalization mechanisms. We propose using scenarios to extract systems requirements, user's task, and related features. We argue that if we clearly identify the personalization-related concerns, we could add a posteriori personalization features to legacy systems, following an advance separation of concerns such as aspect, subject and so on.

Introduction

There have been some attempts to identify aspects at the early stages of the software life cycle [Tekinerdogan et al., 1998; Grundy, 2000]. These approaches aim to achieve some of the desired software engineering quality factors, such as reducing adaptation, evolution and maintenance costs. In this position paper, we are interested in identifying early aspects in personalization systems.

It is a known fact that the Internet has brought about deep changes in the way people work and in the way software designers conceive and develop computational artifacts. One interesting evolution is related to the adaptation mechanisms involved in personalization. In a nutshell, “personalization is about building customer loyalty by building a meaningful one-to-one relationship; by understanding the needs of each individual and helping satisfy a goal that efficiently and knowledgeably addresses each individual's need in a given context (...) it is about the mapping and satisfying of a user's/customer's goal in a specific context with a service's/business's goal in its respective context” [Riecken, 2000]. Although e-commerce has been the driving force underlying the increase in personalization features, many applications profit from the ongoing research and technological advances in this area. We adopt here Blom's definition of personalization, as “a process that changes the functionality, interface, information content, or distinctiveness of a system to increase its personal relevance to an individual.” [Blom, 2000].

The purpose of this position paper is to suggest that advanced separation of concerns (AsoC) approaches may help specify personalization solution and point further investigation about whether personalization-related concerns may be identified early in

the development process and, if so, how it may impact software design, flexibility and maintainability. In this position paper, we will utilize scenarios of use that depict some personalization characteristics as a resource to try to extract concerns as early aspects.

If we are able to clearly identify personalization-related concerns, we will be able to assess the feasibility of adding *a posteriori* personalization features to existing applications, following an advanced separation of concerns approach such as aspect-oriented programming [Kiczales et al, 1997], subject-oriented programming [Harrison et al, 1994], composition filters [Aksit et al, 1994] and so on.

Human-Computer Interaction and Personalization

Many concerns underlying personalization are directly related to the research area of human-computer interaction (HCI). Design of personalization artifacts should be driven by users' goals and needs, not by the novelty of cool tools [Kramer et al., 2000]. Karat et al. put it nicely: "Every tool should feel like it was custom designed for you, the user in your context." [Karat et al., 2000]. Usability evaluation methods and tools have traditionally assessed the quality of use of an interactive system. Usability issues may be thought of as cross-cutting concerns which drive design. A typical set of usability concerns is as follows [Nielsen, 1993]:

- ease of learning and understanding: How long does it take for users to learn how to use an application? After a period of not using an application, can users still remember how it works?
- ease of use: Does the application support the user's tasks adequately? Are there many possibilities of making mistakes? Does the application provide adequate and timely feedback?
- user control: Can the users predict what will happen when they trigger an action? Can they interrupt lengthy processes?
- flexibility: Are there alternative ways to perform a task and achieve a goal?
- efficiency: How long does it take to perform a task? How long does it take to recover from an error? Does the application succeed in improving the user's productivity?

These concerns are interrelated, and design decisions usually involve cost/benefit analysis and tradeoffs. For instance, applications for highly specialized users may provide ease of use at the expense of more extensive training needs. Flexible applications may provide both ease of use and ease of learning, possibly by means of two or more distinct interaction paths leading to the same result, i.e., achieving the same goal.

How usability concerns affect the resulting design and implementation and propagate throughout design is the focus of HCI research. We will address in this paper the identification of concerns as related specifically to personalization mechanisms, and leave the relationship between these concerns and HCI concerns to future research. Filman has stated that AOP is useful for handling systematic requirements/concerns¹

¹ " Systematic concerns/requiments pervade the behavior of the system, but can be realized by 'doing the right thing' in 'all the right places'" [Filman, 1998].

[Filman, 2001]. An interesting issue would be whether (and if so, which) usability concerns might be considered systematic concerns.

Domain-Separable Analysis of Personalization

In our first attempt to identify personalization concerns, we have found three levels of concerns that may be analyzed independently of problem domain, i.e., as a solution domain.

- **syntactic level** (domain-independent): those mechanisms that do not involve domain information, but only users' (as a whole) activity in the application. For instance, the list of the n most recent items viewed or inserted, the list of n most visited items, and so on;
- **semantic level** (domain- and user-dependent, but task-independent): those mechanisms that make use of relations between entities in the domain, such as those represented in a domain ontology, taking into account user information. For example, the list of items related to the user's (as an individual) interests;
- **pragmatic level** (domain-, user- and task-dependent): those mechanisms that take into account information which not only has a correspondence to the conceptual domain, but also to the users' tasks. For instance, the user's agenda of tasks that should be carried out at the moment, a list of tasks that are late, or a list of users whose tasks are affecting the current user's work.

Later stages in the design will need to take into account domain-specific information, but in our research we will try to keep our analysis domain-independent, considering two complementary views of an application: a content view and a process view.

In the **content view**, users' goals are not task-related, but content-related. In other words, users need only to find some information, by whichever means available in the application. This view represents the portions of an application designed for information-retrieval, for instance. In the **process view**, the focus are the tasks users need to perform using the application. The content is important only inasmuch as it affects the task at hand. This view is the predominant view of workflow-related systems.

The content view may present personalization at both syntactic and semantic levels, but not at the pragmatic level. This view has no representation of users' tasks. The process view, on the other hand, is more comprehensive, in that it includes both task and content representations. Due to lack of space we won't be able to address both of them here. We will present only a simple scenario related to the content view.

Sample Scenarios

To help identify concerns early in the design process, we propose using scenarios [Carroll, 1995], from which we may extract systems requirements and users tasks and related features. We present a scenario that illustrates possible usage situations of applications without personalization, and suggest some directions for adding personalization features to them, while discussing the corresponding concerns.

Scenario I: Information retrieval application

John needs to find a book to study for his final exams on Non-Euclidean Geometry next week. He enters the university library website. Since the book is dated 10 years ago, he doesn't check the list of recent acquisitions. He proceeds to browse the index of Mathematics and is led to another index. He notices it may take a while to find the book that way and, since he knows the name of the book he is looking for, he decides to search the book by providing the author's last name. The system presents summary information of 12 books, and he quickly locates the book he wants. He examines the book details, in order to see in which section in the library the book is located. He exits the system and rushes to the library to get the book.

In this typical information retrieval system, we identify, by means of the scenario, the following features: *browsing through a structure of indexes*, *searching a database*, *looking into a list of highlighted items* (in this case, recent acquisitions). All of the aforementioned usability concerns are at play here: ease of learning (the user shouldn't need training); ease of use (the user shouldn't make mistakes because of bad interface design); user control (the user may change his/her mind during interaction and interrupt the current task); flexibility (there are alternative ways to reach a goal); and efficiency (the user finds information faster by using the system than without it).

Without personalization, every user is presented with the same content and structure. The following scenario illustrates a simple feature addition to the system in order to provide a small degree of personalization:

John has decided to do some research on Software Engineering, and needs to find a few books that may be important for his work. The first book he finds is a classic, and he must read it. Fortunately, the new version of the library system presents a mechanism for *bookmarking* the books he finds interesting, and John decides to try this new feature right away. He looks up 8 or 9 books about Software Engineering, 3 of which he decides to bookmark. He checks the items that have been bookmarked, and decides to start from "the classic". He examines the information of that book's location in the library, exits the system and goes to the library to check that book out.

With such a customization mechanism, the library homepage can supply a section called something like "my favorite books", which presents a list of books that have been bookmarked by the current user. One should notice that, by adding this bookmarking feature, another user goal emerges: the user will need now to be able to manage his/her bookmarks.

Upon inspecting the scenario, we have identified the following requirements:

- user control of the adaptation: in customization mechanisms such as the one presented here, the user has total control of the adaptation. In terms of Kühme's classification [Kühme et al. 1993], the user is responsible for initiating the adaptation process,

deciding whether or not to adapt, and executing the adaptation (adding the current item to the bookmark list)

- persistence: the information of this session should be stored for later recall in future sessions with the application
- user model (user's bookmarks): the personalized information is related only to the individual user responsible for the customization

Another kind of personalization is via recommendations, where a list of items considered important is presented to the user at certain points during the interaction. This list may be inferred by the system based on the user's profile, his/her behavior during interaction with the system, and even on other (somehow correlated) users' profiles. In our library example, the system could infer a set of relevant books based on the user's interaction history over the last few sessions.

The following requirements are identified in this part of the scenario:

- system control of the adaptation: the system takes initiative, decides and executes the adaptation, which corresponds to the creation and maintenance of the list of recommended items
- monitoring: the system monitors the users' activities within the system
- user model (interaction history): the individual user's activities within the system provide input to the personalization mechanism
- persistence: the information of this session should be stored for later recall in future sessions with the application

Personalization mechanisms are typically scattered in software design. These mechanisms involve some concerns already identified as suitable for AOSD: log history, monitoring, tracking, and persistence, to name a few. We believe personalization may be analyzed and specified using ASoC approaches [TRESE, 2002].

Conclusions and ongoing work

As software evolves, new concerns may arise. In this position paper, we have taken a first step towards an analysis about whether early aspects will improve the ability to "remodularize" software according to new concerns in a non-invasive way and without eliminating existing encapsulation, i.e., encapsulation derived from previously identified concerns.

An issue that deserves further investigation is whether the concerns we have identified as crosscutting concerns at the requirements stage would be propagated as an aspect throughout design, i.e., if the aspectual composition would be maintained at the design and implementation levels.

As for the early stages of requirements analysis and design, we need robust methods to help identify crosscutting concerns from scenarios, as well as the relationships between these concerns.

With respect to personalization systems, we need to develop a real case study to investigate how the identification of early aspects might affect both the design process and the resulting design. We will create scenarios of use that depict more sophisticated personalization mechanisms, and scrutinize them carefully, in order to derive representations of users' tasks and system requirements, together with the corresponding concerns, if any. Having captured this information, we will investigate how the identified concerns will propagate throughout the design process.

Acknowledgments

Our thanks to Christina von Flach and Alessandro Garcia for comments to drafts this position paper.

References

- Aksit et al., 1994 M. Aksit and K. Wakita and J. Bosch and L. Bergmans, Abstracting object interactions using composition filters, Lecture Notes in Computer Science, 1994.
- Blom, 2000 Blom, J. "Personalization – a taxonomy". In *Proceedings of the CHI 2000 Workshop on Designing Interactive Systems for 1-to-1 E-commerce*. Online at <http://www.zurich.ibm.com/~mrs/chi2000/> (last accessed Mar/2002).
- Carroll, 1995 Carroll, J. (ed) *Scenario-based Design: Envisioning Work and Technology in System Development*. New York, NY. John Wiley and Sons, 1995.
- Grundy, 2000 Grundy, J.C. Multi-perspective specification, design and implementation of components using aspects, International Journal of Software Engineering and Knowledge Engineering, Vol. 10, No. 6, December 2000, World Scientific
- Harrison et al, 1994 Harold Ossher, William Harrison, Frank Budinsky, and Ian Simmonds, *Subject-Oriented Programming: Supporting Decentralized Development of Objects*, Proceedings of the 7th IBM Conference on Object-Oriented Technology, July, 1994
- Karat et al., 2000 Karat, J.; Karat, C-M.; Ukelson, J. "Affordances, Motivation, and the Design of User Interfaces". In *Communications of the ACM* 43, 8. August, 2000. pp.49–51.
- Kiczales et al., 1997 G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-Oriented Programming. In proceedings of ECOOP '97, Springer-Verlag LNCS 1241. June 1997
- Kramer et al., 2000 Kramer, J.; Noronha, S.; Vergo, J. "A User-Centered Design Approach to Personalization". In *Communications of the ACM*, 8. August, 2000. pp.45–48.
- Kühme et al. 1993 T. Kühme and M. Schneider-Hufschmidt. Introduction. In M. Schneider-Hufschmidt, T. Kühme and U. Malinowski. *Adaptive*

- User Interfaces: Principles and Practice*. North-Holland, 1993, pp.1-9.
- Nielsen, 1993 Nielsen, J. *Usability Engineering*. Academic Press, 1993.
- Riecken, 2000 Riecken, D. “Personalized Views of Personalization”. In *Communications of the ACM* **43**, 8. August, 2000. pp.27–28.
- Tekinerdogan et al., 1998 B. Tekinerdogan and M. Aksit, Deriving *Design Aspects from Canonical Models*, in *Object-Oriented Technology*, S. Demeyer and J. Bosch (Eds.), LNCS 1543, ECOOP'98 Workshop Reader, Springer Verlag, pp. 410-413, July 1998.
- Trese, 2002 TRESE Aspects and advanced separation of concerns homepage. http://trese.cs.utwente.nl/aspects_asoc/index.htm (last accessed Feb/2002).

On Objects, Aspects, and Specifications Addressing their Collaboration

Tommi Mikkonen

Tampere University of Technology, P.O.Box 553, FIN-33101 Tampere, Finland

tjm@cs.tut.fi

ABSTRACT

Aspect-oriented programming enables addressing of cross-cutting concerns in a modular fashion. However, for utilizing that type of modularity effectively, we need to design architectures that enable well-considered use of aspects. This calls for new types of methodologies that enable the treatment of both aspects and objects as first-class citizens, instead of currently advocated approaches like UML that primarily place the focus on object-oriented architecture only, leaving aspects only a role as add-on facilities. In this paper, we sketch an approach where the relation of aspects and objects is defined already in the specification phase, thus allowing balancing between their role in the design. The approach is derived from experiences gained with the specification language DisCo, which was originally targeted as a formal specification method for reactive systems.

1. INTRODUCTION

Aspect-oriented programming enables addressing of cross-cutting concerns in a modular fashion [4]. However, for expressing them in AspectJ language [14], for instance, one needs to compose a completed architecture for the primary design goals first, because aspects are attached to an object-oriented design as a secondary dimension of concerns. The primary design goals should be satisfied with conventional specification notations like UML [16], on top of which aspects are attached. In contrast, hyperslices [11] first require an existing architecture used as a reference for composing different aspects, which can then be given separately.

A consequence of the above approaches is that architecting of systems consisting of collaborating aspects and objects is hard. Moreover, applying iterative and incremental development approaches (see e.g. [6]) that are currently considered favorable is hard. Such practices implicitly assume that software systems can be designed one set of features at a time, whereas managing the features in codesigns of objects and aspects is not supported. While the use of aspects as such enables a closer relation between features and code than conventional approaches, designing a new increment that does not interfere with already existing software is not eased with aspects in the general case. In fact, mastering the interaction of cross-cutting

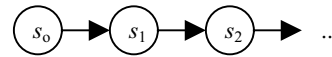


Figure 1. Execution as a state sequence

concerns and conventional code can be harder than if only traditional approaches had been used. To overcome the above problems, we need facilities that enable enhanced facilities for specifying the relation of aspects and conventional objects.

The rest of this paper is structured as follows. Section 2 discusses specification of software architectures in general, and provides an introduction on the different dimensions of software architectures that address the use of both objects and aspects. Section 3 introduces the specification method DisCo [15], and lists practical experiences on aspect-oriented modeling and specification gained with it. Section 4 finally concludes the paper with some final remarks.

2. SPECIFYING AN ARCHITECTURE

Software architecture is the first thing to be fixed when developing a software system [13]. Describing an architecture means construction of an abstract model that exhibits certain kinds of intended properties. Such a model is *operational*, if it formalizes executions as state sequences, as illustrated in Figure 1. In the figure, all the variables in the model have unique values in each state s_i .

2.1 Conventional Architecture Modeling

Fundamentally, the purpose of architecting is to partition the intended system into smaller pieces that can be attacked separately. In conventional approaches, the core is to define interfaces that encapsulate internals of modules. With the interfaces remaining unchanged, the underlying implementation can be changed relatively easily to use a more memory-efficient data structure, for instance. The design of behaviors can then be based on scenarios or use cases that denote the sequences needed for executions in terms of method names.

The use of interfaces and scenarios as the semantics of behaviors relies on the expectation that the semantics of an execution can be composed from the semantics of the

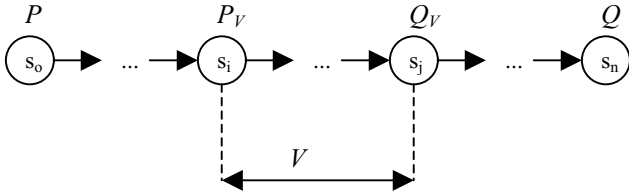


Figure 2. Subsequence in a behavior

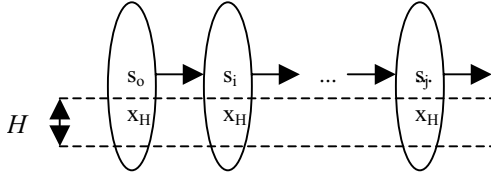


Figure 3. Horizontal projection in a behavior

methods attached to the components in a conventional architecture. More generally, an architecture that consists of conventional units can be seen to impose a structure of nested subsequences on each state sequence. Operations available for addressing the behavior of the system with architectural units are then *sequential composition*, i.e., concatenation of operations of one component, and *invocation*, i.e., embedding of a sequence produced by a component in a longer sequence. For both primitive operations, the resulting state sequences have subsequences for which the components are responsible, as illustrated in Figure 2.

In this paper, we will refer to the architectural dimension represented by this kind of modularity as *vertical*. Vertical architectures provide a natural basis for structuring the responsibility for implementation of different components in conventional systems.

2.2 Addressing cross-cutting concerns

As an alternative for focusing on invoked interface operations, the architecture can be modeled by focusing on how the variables of the system behave in an execution, similarly to program slicing [12]. However, unlike in conventional slicing, we are interested in composing new systems with such projections, not on decomposing existing ones. Such an architecture imposes a slicing or a projection of state sequences, where each unit is responsible for some subset of variables in all phases of the execution. We will refer to this architecture dimension as *horizontal*. The situation is illustrated in Figure 3.

The use of horizontal units makes the generation of state sequences fundamentally different from sequences relying on vertical architectures. The two basic operations between horizontal units are *parallel composition* that merges state sequences generated by the component units, and *superposition* that utilizes some sequences generated by a component unit, embedding them in sequences that

involve a larger set of variables. With both primitive operations, the resulting state sequences define projections for which the horizontal components are responsible.

The two dimensions of an architecture characterized above contrast each other. From the viewpoint of a vertical architecture, the behaviors generated by horizontal units represent crosscutting concerns, and from the horizontal viewpoint vertical units emerge incrementally as horizontal units are put together and embedded in larger units.

Unfortunately, we have no universally accepted approach for composing systems out of horizontal projections. The fundamental problem is that with projections, objects “grow” when more and more projections are introduced. Handling this growing at the level of programming abstraction is difficult, because the mapping of behavioral increments to a system with control flows becomes increasingly complex. Approaches that have an established reputation include aspect-oriented programming as introduced in AspectJ, where the developer tells places for join points in an existing system, frameworks, where the developer is responsible for the correct use of inheritance, and design patterns, where the designer should basically only reuse interfaces and ideas of collaboration. In hyperslices, an existing architecture is used to guide the composition.

We argue that the use of the horizontal architectural dimension is essential for improved understanding of software, as already demonstrated by the use of aspects and program slices. However, in order to architect with the horizontal dimension, we must overcome problems of composition, interference, and control flows. To accomplish this, the level of abstraction must be raised. The price is that while the systems remain executable, there may not be a direct mapping to program code.

3. USING HORIZONTAL DIMENSION

Our experiences on architecting with horizontal units of architecture arise from the design and use of the specification method DisCo [15]. In the following, we will provide a short introduction to the method.

3.1 Introduction of the DisCo method

The two basic items of every DisCo specification are objects and actions. Objects are instances of classes. For example, the following class definition introduces a class that contains one instance variable x of type integer.

```
class myClass = { x : integer };
-- Instances of this class contain instance
-- variable x.
```

Actions are used to mutate the states of objects involved in an execution. Each action has a signature that contains a list of objects needed for an execution, an enabling guard, and a list of assignments to the variables of involved objects. Thus, actions can be taken as multi-object

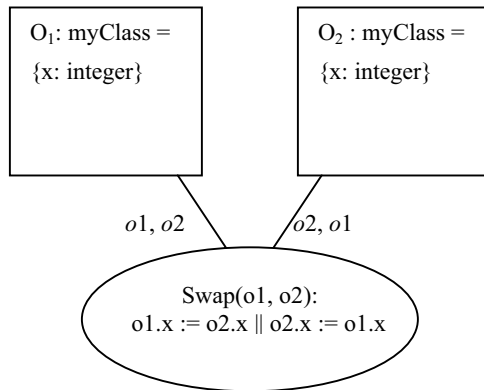


Figure 4. Simple action and two objects

methods. The following excerpt introduces a simple action that swaps the values of two instances of class *myClass*. In the action, Ada-style comments are used for explaining its structure.

```

action Swap(o1, o2 : myClass) is
-- Signature (participating objects,
--           their types and roles)
when true do
-- Enabling condition
    o1.x := o2.x || o2.x := o1.x;
-- Changes in states of participants
end Swap;
  
```

Figure 4 gives an illustration of two simple objects (boxes) and one simple action with two parallel assignments (ellipse) given above. Participant roles in the action are also included in the figure. The execution of actions is atomic, i.e., once started, executions are bound to be finished without interference. Executions take place in an interleaving fashion. The action to be executed next is selected nondeterministically from those whose enabling condition is true. Therefore, problems related to control flows need not be considered except when such restrictions are explicitly included in an action-based model.

Specifications given with actions and objects, i.e., instantiated classes, are used for generating operational interpretations. This has allowed us to introduce tool support for executing completed specifications [1].

3.2 Incrementality

For modularity, the main mechanism adopted in the DisCo approach is superposition. As such, superposition is a well-known technique whose usage was first reported in [3]. However, its close relation to aspect-orientation was pointed out only more recently by Katz and Gil [8].

In DisCo, superposition is used as follows. Each module of the specification is given as a *layer* that adds variables and

related operations to existing systems. Layers are horizontal. In other words, they are allowed to add details to multiple classes and actions, which make objects given in the specification grow.

When using superposition in connection with a joint action, new participants, restrictions on executions, and new assignments can be added, but they can never violate the original action. Logically, this means that any action must imply all the actions it has been derived from with superposition. In practice, the only restriction for new or modified actions is that it is only possible to modify the values of variables given in the same layer. For instance, extending the previously given class with a variable denoting whether or not the system is turned on or not would yield the following extensions. In the listing, three dots (...) are used to refer to the old parts of actions.

```

class myClass =
    myClass + {power: Boolean};
-- Adds a new variable to the class

refined Swap is
-- Adds new issues to an existing action
when ... o1.power and o2.power do
-- Makes the enabling condition stronger
    ...
-- No new state changes
end Swap;
  
```

In addition, an action is needed for turning the power on and off,

```

action OnOff(o:myClass) is -- New action
when true do
    o.power := not o.power;
-- Only new variables are assigned to
end OnOff;
  
```

Figure 5 extends the previous figure with the above additions. The contents of the new layer are included in the figure with italics, indicating the additions of the new layer in a complete system.

3.3 Relation to aspects and hyperslices

In the above example, it is essential that while the effect of the new layer is cross-cutting, its syntactic representation remains modular. Superposition steps are the building blocks for specifications, and, vice versa, specifications can be projected to individual layers where superposition has been used. Therefore, with each layer applying

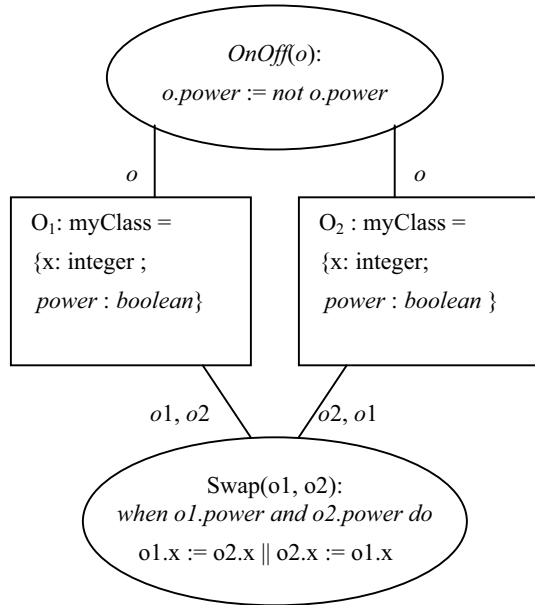


Figure 5. Introducing a new layer

superposition, it is possible to compose separately given layers into a more complex specification. Practice has shown that a common way to give specifications is to have one common ancestor specification that gives the most general specification for a system. Then, different branches introduce new properties in an aspect-oriented fashion. In the end, all the branches are composed into a complete specification, if all the layers introduce disjoint sets of new variables. This results in a true aspect-oriented specification style, where different aspects are given separately with superposition acting as the guarantee for later composability. Effectively, this approach provides facilities that are similar to hyperslices [11] for specification, with each branch representing a different hyperslice, with enforced composability.

Figure 6 gives an example of this approach. The figure illustrates the specification architecture of a telecom exchange modeled with DisCo. Each superposition and composition operations and resulting specifications have been explicitly depicted. Each of the different branches (or hyperslice) has an effect on e.g. on how call control takes place, which finally becomes formalized completely in specification *Completed Connections*. Being able to give the different concerns separately makes it easier to understand their eventual relation in the implementation. Figure 7 lists the attributes of class *Connection* derived in the example mentioned above. The contribution of the different layers is denoted with comments in more detail. In addition to the variables in the class, the DisCo specification also gives a definition for the behavior in a modular fashion.

Published results have confirmed that the DisCo approach enables formal specification of combinations of patterns

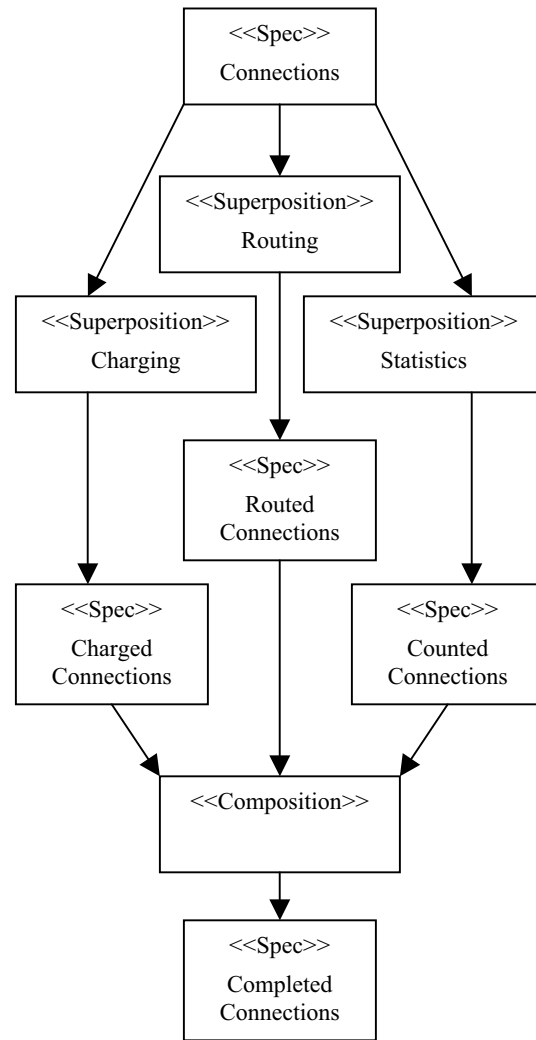


Figure 6. Specification architecture of a telephone

[9]. As an example, we derived a specification of a system utilizing both Mediator and Observer patterns of [5], with formal justification provided for the interaction of the patterns. Moreover, the approach has also been used in aspect-oriented specification of a real-time system in [7]. In that context, real time was treated as a separate aspect at the specification level. The approach has also been used for modeling and specification of increments delivered in different releases [10] and for the management of software evolution [2], which both benefit from advanced separation of concerns. Notice however that the level of DisCo specifications is abstract, and additional design decisions are needed for a practical implementation. Of course, as the specification contains both objects and aspects as separable entities, it is easier to partition the specification into objects and aspects for implementation.

4. CONCLUSIONS

In order to use both aspects and objects already in the specification phase, we need a methodology that considers both aspects and objects as first-class citizens. As long as

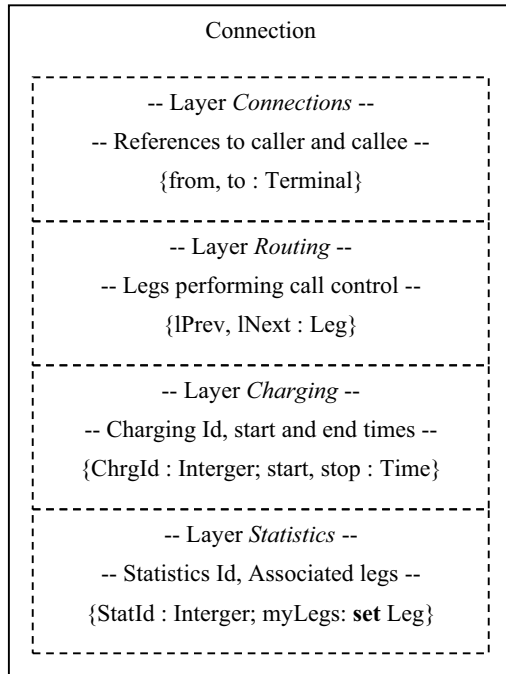


Figure 7. Class *Connection* in the specification

we consider aspects as add-on features to already complete systems, we do not intentionally design systems so that a certain efficient aspect implementation would be an option. Therefore, we are bound to use aspects inefficiently, in ad-hoc fashion, and for secondary features only.

In this paper, we sketched an approach that is capable of handling the relation between objects and aspects at an abstract level. The driving force behind the introduced specification approach is the use of superposition as a basis for modularity instead of conventionally used invocation-based relations. This allows specifications where both objects and aspects are addressed in a collaborative fashion. Moreover, the focus can be placed on both objects and aspects at the same time.

In connection with conventionally accepted aspect-oriented approaches, the proposed method relates as follows. The way we separate between logically different issues reminds hyperslices in the sense that the parts that are relevant for some concern are addressed in a modular fashion. At the same time, individual modules that the specifier gives remind aspects of AspectJ. We argue that this combination, in connection with raised level of abstraction and associated tools, provides a natural basis for specifications addressing objects, aspects, and their collaboration.

REFERENCES

- [1] Aaltonen, T., Katara, M. and Pitkänen, R. DisCo toolset – the new generation. *Journal of Universal Computer Science*, 7(1):3-18, 2001.
- [2] Aaltonen, T. and Mikkonen, T. Managing software evolution with a formalized abstraction hierarchy. *Proc. International Workshop on Formal Foundations of Software Evolution*, Lisbon, Portugal, March 2001.
- [3] Dijkstra, E. W. and Scholten, C. S. Termination detection for diffusing computations. *Information Processing Letters*, 11(1), 1-4, 1980.
- [4] Elrad T., Filman, R.E. and Bader, E. Aspect-oriented programming. *Communications of the ACM*, 11(1): 29-32, October 2001.
- [5] Gamma, E., Helm, R., Johnson, R., and Vlissides J. *Design Patterns*. Addison Wesley, Reading, MA, 1995.
- [6] Jacobson, I. Booch, G. and Rumbaugh, J. *The Unified Software Development Process*. Addison Wesley, Reading, MA, 1999.
- [7] Katara M. and Mikkonen, T. Aspect-oriented specification architectures for distributed real-time systems. 180-190, *Proceedings of the Seventh IEEE International Conference on Engineering of Complex Systems (Eds. S.F. Andler, M.G. Hinchey, and J. Offutt)*, IEEE Computer Society Press, 2001.
- [8] Katz, S. and Gil, J. Aspects and superimpositions. *Position paper in Aspect Oriented programming workshop in ECOOP'99*, Lisbon, Portugal, June 1999.
- [9] Mikkonen, T. Formalizing design patterns. 115-124, *Proceedings of the 1998 International Conference on Software Engineering*, IEEE Computer Society, 1998.
- [10] Mikkonen, T. and Järvinen, H.-M. Specifying for releases. *International Workshop on Principles of Software Evolution*, 118-122, April 20-21, Kyoto, Japan, 1998.
- [11] Ossher, H. and Tarr, P. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 11(1): 43-50, October 2001.
- [12] Weiser, M. Program slicing. *IEEE Transactions on Software Engineering*, 352-357, 10(4), 1984.
- [13] IEEE recommended practice for architecture description. *IEEE std 1471*, 2000.
- [14] AspectJ homepage. At URL <http://aspectj.org>.
- [15] DisCo project homepage. At URL <http://disco.cs.tut.fi>.
- [16] UML homepage. At URL <http://www.rational.com/uml>

On Objects, Aspects, and Specifications Addressing their Collaboration

Tommi Mikkonen

Tampere University of Technology, P.O.Box 553, FIN-33101 Tampere, Finland

tjm@cs.tut.fi

ABSTRACT

Aspect-oriented programming enables addressing of cross-cutting concerns in a modular fashion. However, for utilizing that type of modularity effectively, we need to design architectures that enable well-considered use of aspects. This calls for new types of methodologies that enable the treatment of both aspects and objects as first-class citizens, instead of currently advocated approaches like UML that primarily place the focus on object-oriented architecture only, leaving aspects only a role as add-on facilities. In this paper, we sketch an approach where the relation of aspects and objects is defined already in the specification phase, thus allowing balancing between their role in the design. The approach is derived from experiences gained with the specification language DisCo, which was originally targeted as a formal specification method for reactive systems.

1. INTRODUCTION

Aspect-oriented programming enables addressing of cross-cutting concerns in a modular fashion [4]. However, for expressing them in AspectJ language [14], for instance, one needs to compose a completed architecture for the primary design goals first, because aspects are attached to an object-oriented design as a secondary dimension of concerns. The primary design goals should be satisfied with conventional specification notations like UML [16], on top of which aspects are attached. In contrast, hyperslices [11] first require an existing architecture used as a reference for composing different aspects, which can then be given separately.

A consequence of the above approaches is that architecting of systems consisting of collaborating aspects and objects is hard. Moreover, applying iterative and incremental development approaches (see e.g. [6]) that are currently considered favorable is hard. Such practices implicitly assume that software systems can be designed one set of features at a time, whereas managing the features in codesigns of objects and aspects is not supported. While the use of aspects as such enables a closer relation between features and code than conventional approaches, designing a new increment that does not interfere with already existing software is not eased with aspects in the general case. In fact, mastering the interaction of cross-cutting

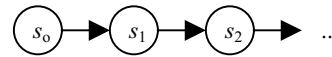


Figure 1. Execution as a state sequence

concerns and conventional code can be harder than if only traditional approaches had been used. To overcome the above problems, we need facilities that enable enhanced facilities for specifying the relation of aspects and conventional objects.

The rest of this paper is structured as follows. Section 2 discusses specification of software architectures in general, and provides an introduction on the different dimensions of software architectures that address the use of both objects and aspects. Section 3 introduces the specification method DisCo [15], and lists practical experiences on aspect-oriented modeling and specification gained with it. Section 4 finally concludes the paper with some final remarks.

2. SPECIFYING AN ARCHITECTURE

Software architecture is the first thing to be fixed when developing a software system [13]. Describing an architecture means construction of an abstract model that exhibits certain kinds of intended properties. Such a model is *operational*, if it formalizes executions as state sequences, as illustrated in Figure 1. In the figure, all the variables in the model have unique values in each state s_i .

2.1 Conventional Architecture Modeling

Fundamentally, the purpose of architecting is to partition the intended system into smaller pieces that can be attacked separately. In conventional approaches, the core is to define interfaces that encapsulate internals of modules. With the interfaces remaining unchanged, the underlying implementation can be changed relatively easily to use a more memory-efficient data structure, for instance. The design of behaviors can then be based on scenarios or use cases that denote the sequences needed for executions in terms of method names.

The use of interfaces and scenarios as the semantics of behaviors relies on the expectation that the semantics of an execution can be composed from the semantics of the

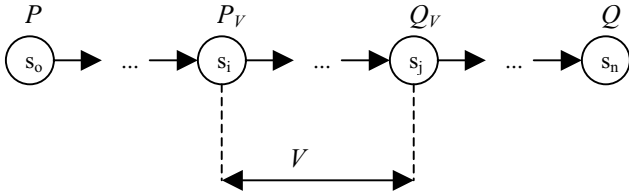


Figure 2. Subsequence in a behavior

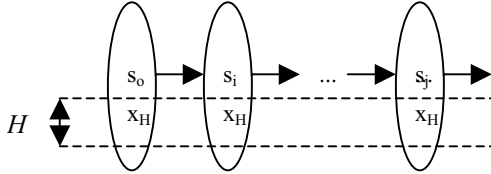


Figure 3. Horizontal projection in a behavior

methods attached to the components in a conventional architecture. More generally, an architecture that consists of conventional units can be seen to impose a structure of nested subsequences on each state sequence. Operations available for addressing the behavior of the system with architectural units are then *sequential composition*, i.e., concatenation of operations of one component, and *invocation*, i.e., embedding of a sequence produced by a component in a longer sequence. For both primitive operations, the resulting state sequences have subsequences for which the components are responsible, as illustrated in Figure 2.

In this paper, we will refer to the architectural dimension represented by this kind of modularity as *vertical*. Vertical architectures provide a natural basis for structuring the responsibility for implementation of different components in conventional systems.

2.2 Addressing cross-cutting concerns

As an alternative for focusing on invoked interface operations, the architecture can be modeled by focusing on how the variables of the system behave in an execution, similarly to program slicing [12]. However, unlike in conventional slicing, we are interested in composing new systems with such projections, not on decomposing existing ones. Such an architecture imposes a slicing or a projection of state sequences, where each unit is responsible for some subset of variables in all phases of the execution. We will refer to this architecture dimension as *horizontal*. The situation is illustrated in Figure 3.

The use of horizontal units makes the generation of state sequences fundamentally different from sequences relying on vertical architectures. The two basic operations between horizontal units are *parallel composition* that merges state sequences generated by the component units, and *superposition* that utilizes some sequences generated by a component unit, embedding them in sequences that

involve a larger set of variables. With both primitive operations, the resulting state sequences define projections for which the horizontal components are responsible.

The two dimensions of an architecture characterized above contrast each other. From the viewpoint of a vertical architecture, the behaviors generated by horizontal units represent crosscutting concerns, and from the horizontal viewpoint vertical units emerge incrementally as horizontal units are put together and embedded in larger units.

Unfortunately, we have no universally accepted approach for composing systems out of horizontal projections. The fundamental problem is that with projections, objects “grow” when more and more projections are introduced. Handling this growing at the level of programming abstraction is difficult, because the mapping of behavioral increments to a system with control flows becomes increasingly complex. Approaches that have an established reputation include aspect-oriented programming as introduced in AspectJ, where the developer tells places for join points in an existing system, frameworks, where the developer is responsible for the correct use of inheritance, and design patterns, where the designer should basically only reuse interfaces and ideas of collaboration. In hyperslices, an existing architecture is used to guide the composition.

We argue that the use of the horizontal architectural dimension is essential for improved understanding of software, as already demonstrated by the use of aspects and program slices. However, in order to architect with the horizontal dimension, we must overcome problems of composition, interference, and control flows. To accomplish this, the level of abstraction must be raised. The price is that while the systems remain executable, there may not be a direct mapping to program code.

3. USING HORIZONTAL DIMENSION

Our experiences on architecting with horizontal units of architecture arise from the design and use of the specification method DisCo [15]. In the following, we will provide a short introduction to the method.

3.1 Introduction of the DisCo method

The two basic items of every DisCo specification are objects and actions. Objects are instances of classes. For example, the following class definition introduces a class that contains one instance variable x of type integer.

```
class myClass = { x : integer };
-- Instances of this class contain instance
-- variable x.
```

Actions are used to mutate the states of objects involved in an execution. Each action has a signature that contains a list of objects needed for an execution, an enabling guard, and a list of assignments to the variables of involved objects. Thus, actions can be taken as multi-object

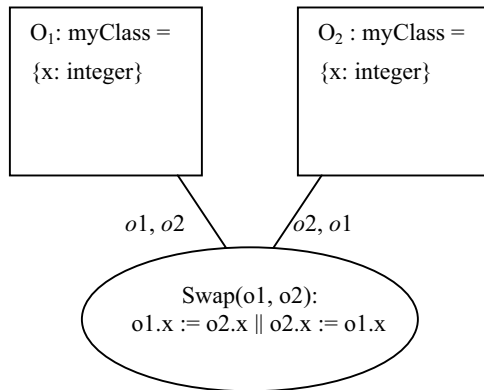


Figure 4. Simple action and two objects

methods. The following excerpt introduces a simple action that swaps the values of two instances of class *myClass*. In the action, Ada-style comments are used for explaining its structure.

```

action Swap(o1, o2 : myClass) is
-- Signature (participating objects,
--           their types and roles)
when true do
-- Enabling condition
    o1.x := o2.x || o2.x := o1.x;
-- Changes in states of participants
end Swap;
  
```

Figure 4 gives an illustration of two simple objects (boxes) and one simple action with two parallel assignments (ellipse) given above. Participant roles in the action are also included in the figure. The execution of actions is atomic, i.e., once started, executions are bound to be finished without interference. Executions take place in an interleaving fashion. The action to be executed next is selected nondeterministically from those whose enabling condition is true. Therefore, problems related to control flows need not be considered except when such restrictions are explicitly included in an action-based model.

Specifications given with actions and objects, i.e., instantiated classes, are used for generating operational interpretations. This has allowed us to introduce tool support for executing completed specifications [1].

3.2 Incrementality

For modularity, the main mechanism adopted in the DisCo approach is superposition. As such, superposition is a well-known technique whose usage was first reported in [3]. However, its close relation to aspect-orientation was pointed out only more recently by Katz and Gil [8].

In DisCo, superposition is used as follows. Each module of the specification is given as a *layer* that adds variables and

related operations to existing systems. Layers are horizontal. In other words, they are allowed to add details to multiple classes and actions, which make objects given in the specification grow.

When using superposition in connection with a joint action, new participants, restrictions on executions, and new assignments can be added, but they can never violate the original action. Logically, this means that any action must imply all the actions it has been derived from with superposition. In practice, the only restriction for new or modified actions is that it is only possible to modify the values of variables given in the same layer. For instance, extending the previously given class with a variable denoting whether or not the system is turned on or not would yield the following extensions. In the listing, three dots (...) are used to refer to the old parts of actions.

```

class myClass =
    myClass + {power: Boolean};
-- Adds a new variable to the class

refined Swap is
-- Adds new issues to an existing action
when ... o1.power and o2.power do
-- Makes the enabling condition stronger
    ...
-- No new state changes
end Swap;
  
```

In addition, an action is needed for turning the power on and off,

```

action OnOff(o:myClass) is -- New action
when true do
    o.power := not o.power;
-- Only new variables are assigned to
end OnOff;
  
```

Figure 5 extends the previous figure with the above additions. The contents of the new layer are included in the figure with italics, indicating the additions of the new layer in a complete system.

3.3 Relation to aspects and hyperslices

In the above example, it is essential that while the effect of the new layer is cross-cutting, its syntactic representation remains modular. Superposition steps are the building blocks for specifications, and, vice versa, specifications can be projected to individual layers where superposition has been used. Therefore, with each layer applying

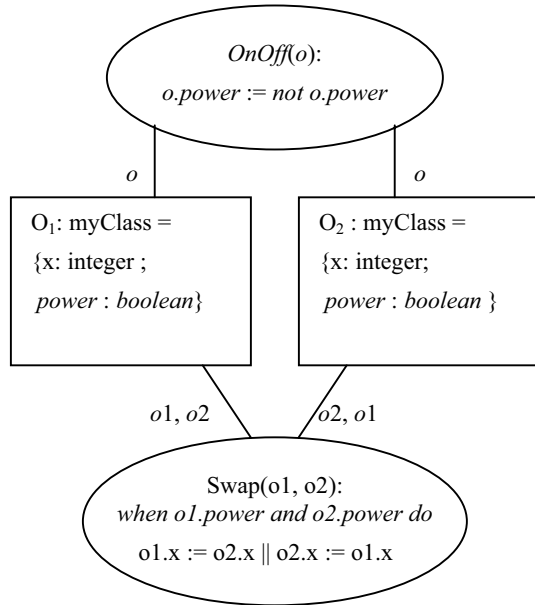


Figure 5. Introducing a new layer

superposition, it is possible to compose separately given layers into a more complex specification. Practice has shown that a common way to give specifications is to have one common ancestor specification that gives the most general specification for a system. Then, different branches introduce new properties in an aspect-oriented fashion. In the end, all the branches are composed into a complete specification, if all the layers introduce disjoint sets of new variables. This results in a true aspect-oriented specification style, where different aspects are given separately with superposition acting as the guarantee for later composability. Effectively, this approach provides facilities that are similar to hyperslices [11] for specification, with each branch representing a different hyperslice, with enforced composability.

Figure 6 gives an example of this approach. The figure illustrates the specification architecture of a telecom exchange modeled with DisCo. Each superposition and composition operations and resulting specifications have been explicitly depicted. Each of the different branches (or hyperslice) has an effect on e.g. on how call control takes place, which finally becomes formalized completely in specification *Completed Connections*. Being able to give the different concerns separately makes it easier to understand their eventual relation in the implementation. Figure 7 lists the attributes of class *Connection* derived in the example mentioned above. The contribution of the different layers is denoted with comments in more detail. In addition to the variables in the class, the DisCo specification also gives a definition for the behavior in a modular fashion.

Published results have confirmed that the DisCo approach enables formal specification of combinations of patterns

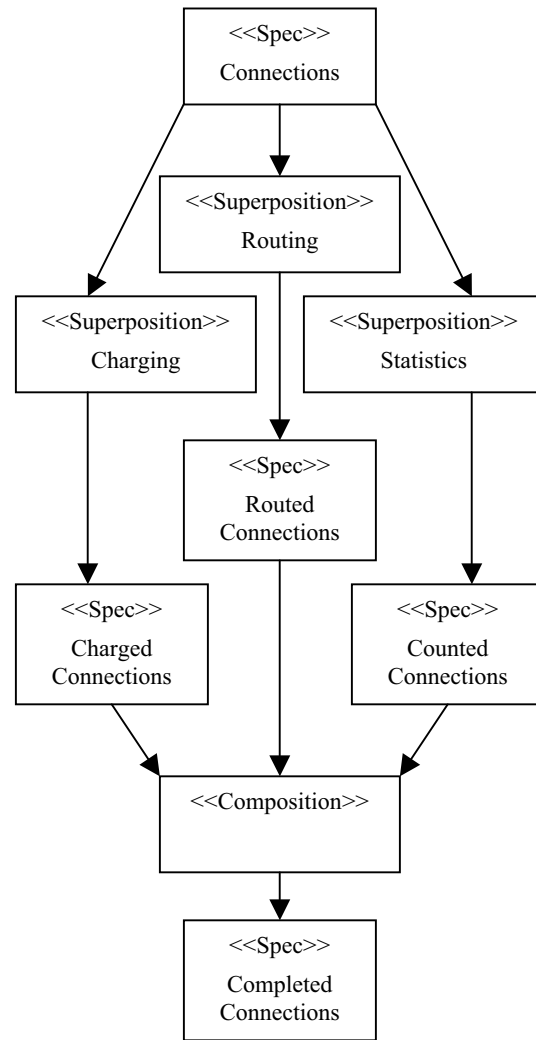


Figure 6. Specification architecture of a telephone

[9]. As an example, we derived a specification of a system utilizing both Mediator and Observer patterns of [5], with formal justification provided for the interaction of the patterns. Moreover, the approach has also been used in aspect-oriented specification of a real-time system in [7]. In that context, real time was treated as a separate aspect at the specification level. The approach has also been used for modeling and specification of increments delivered in different releases [10] and for the management of software evolution [2], which both benefit from advanced separation of concerns. Notice however that the level of DisCo specifications is abstract, and additional design decisions are needed for a practical implementation. Of course, as the specification contains both objects and aspects as separable entities, it is easier to partition the specification into objects and aspects for implementation.

4. CONCLUSIONS

In order to use both aspects and objects already in the specification phase, we need a methodology that considers both aspects and objects as first-class citizens. As long as

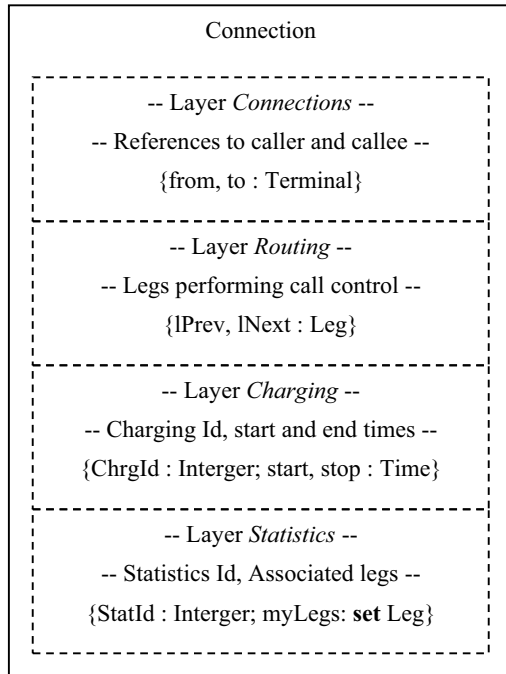


Figure 7. Class *Connection* in the specification

we consider aspects as add-on features to already complete systems, we do not intentionally design systems so that a certain efficient aspect implementation would be an option. Therefore, we are bound to use aspects inefficiently, in ad-hoc fashion, and for secondary features only.

In this paper, we sketched an approach that is capable of handling the relation between objects and aspects at an abstract level. The driving force behind the introduced specification approach is the use of superposition as a basis for modularity instead of conventionally used invocation-based relations. This allows specifications where both objects and aspects are addressed in a collaborative fashion. Moreover, the focus can be placed on both objects and aspects at the same time.

In connection with conventionally accepted aspect-oriented approaches, the proposed method relates as follows. The way we separate between logically different issues reminds hyperslices in the sense that the parts that are relevant for some concern are addressed in a modular fashion. At the same time, individual modules that the specifier gives remind aspects of AspectJ. We argue that this combination, in connection with raised level of abstraction and associated tools, provides a natural basis for specifications addressing objects, aspects, and their collaboration.

REFERENCES

- [1] Aaltonen, T., Katara, M. and Pitkänen, R. DisCo toolset – the new generation. *Journal of Universal Computer Science*, 7(1):3-18, 2001.
- [2] Aaltonen, T. and Mikkonen, T. Managing software evolution with a formalized abstraction hierarchy. *Proc. International Workshop on Formal Foundations of Software Evolution*, Lisbon, Portugal, March 2001.
- [3] Dijkstra, E. W. and Scholten, C. S. Termination detection for diffusing computations. *Information Processing Letters*, 11(1), 1-4, 1980.
- [4] Elrad T., Filman, R.E. and Bader, E. Aspect-oriented programming. *Communications of the ACM*, 11(1): 29-32, October 2001.
- [5] Gamma, E., Helm, R., Johnson, R., and Vlissides J. *Design Patterns*. Addison Wesley, Reading, MA, 1995.
- [6] Jacobson, I. Booch, G. and Rumbaugh, J. *The Unified Software Development Process*. Addison Wesley, Reading, MA, 1999.
- [7] Katara M. and Mikkonen, T. Aspect-oriented specification architectures for distributed real-time systems. 180-190, *Proceedings of the Seventh IEEE International Conference on Engineering of Complex Systems (Eds. S.F. Andler, M.G. Hinchey, and J. Offutt)*, IEEE Computer Society Press, 2001.
- [8] Katz, S. and Gil, J. Aspects and superimpositions. *Position paper in Aspect Oriented programming workshop in ECOOP'99*, Lisbon, Portugal, June 1999.
- [9] Mikkonen, T. Formalizing design patterns. 115-124, *Proceedings of the 1998 International Conference on Software Engineering*, IEEE Computer Society, 1998.
- [10] Mikkonen, T. and Järvinen, H.-M. Specifying for releases. *International Workshop on Principles of Software Evolution*, 118-122, April 20-21, Kyoto, Japan, 1998.
- [11] Ossher, H. and Tarr, P. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 11(1): 43-50, October 2001.
- [12] Weiser, M. Program slicing. *IEEE Transactions on Software Engineering*, 352-357, 10(4), 1984.
- [13] IEEE recommended practice for architecture description. *IEEE std 1471*, 2000.
- [14] AspectJ homepage. At URL <http://aspectj.org>.
- [15] DisCo project homepage. At URL <http://disco.cs.tut.fi>.
- [16] UML homepage. At URL <http://www.rational.com/uml>

Aspects + GAMMA = AspectGAMMA

A Formal Framework for Aspect-Oriented Specification

Mohammad Mousavi¹, Giovanni Russello¹, Michel Chaudron¹, Michel A. Reniers¹, Twan Basten¹, Angelo Corsaro², Sandeep Shukla², Rajesh Gupta², and Douglas C. Schmidt²

¹ Technische Universiteit Eindhoven (TU/e),
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands
{m.r.mousavi,g.russello,m.r.v.chaudron,
m.a.reniers,a.a.basten}@tue.nl

² University of California at Irvine
Irvine, CA 92697

{corsaro@ece, skshukla@ics,rgupta@ics,schmidt@ece}.uci.edu

Abstract. This paper describes an extension to the GAMMA formalism, which we name AspectGAMMA, and we show how non-computational aspects can be expressed separately from the computation in this framework. Examples of such aspects include real-time constraints, location/distribution, behavioral requirements, fault-tolerance, power requirements, and many other aspects. The idea is to abstract the emerging idea of aspect-oriented programming (AOP) into a formal framework, thereby facilitating specification-driven design, enabling formal validation, and design reuse at the requirement specification level. The main goal of this position paper is to outline the way towards a formal foundation of aspect-oriented specification and refinement towards implementation.

1 Introduction

Separation of concerns is one of the concepts at the core of modern software design and evolution. It has been advocated as a key principle for reducing the complexity of developing large-scale software systems [13]. Different techniques and methods have been proposed that help to separate concerns, but some types of concerns, such as timing and distribution, remain hard to separate. These concerns usually *crosscut* the responsibility of several encapsulation units, such as classes in the context of object-oriented programming languages like Java or C++.

One of the fundamental tenets of aspect-oriented programming (AOP)[10] is that complexity in the design of computer programs arises from the fact that many individual concerns (aspects) from the (user) requirements domain are ultimately scattered across multiple locations in the solution (the program). At the same time, however, these parts must be mutually consistent/compatible,

and evolve consistently during further changes [6]. Experience has shown that separating these different concerns explicitly in the software design helps developers manage software complexity more effectively than tangling the concerns into tightly coupled programs. AOP languages, such as AspectJ [15], provide linguistic support for advanced separation of concerns by modeling criteria that correspond to different requirement/design concerns [6].

However, linguistic mechanisms in an implementation language do not help in using the aspect orientation in requirement capturing and other phases of the software engineering lifecycle that precede implementation. As a result, there must be a formal modeling framework that allows requirements to be specified in a formal way that enables the separation of different aspects. To enhance reusability of components, aspect specifications should be independent of each other. The computation itself might also be decomposed into different aspects. This is a basic principle in aspect orientation, the so-called *obliviousness property* [6]. This property suggests that an aspect modeling language should not be aware of other (independent) aspects present in a software architecture. As a result, it should not address the issues of other aspects, to the greatest possible extent. Starting from this assumption, we can deduce that an ideal aspect model would be a simple and tailor-made model that only focuses on a particular well-defined aspect domain. Most of the current requirement specification language/formalism paradigms, such as Object Z [7], however, contain a complex and entangled mixture of different aspects.

This paper discusses the main characteristics of an aspect-oriented formal specification framework, which is based on a multiset transformation language called GAMMA, a formalism based on multiset rewriting [3, 2]. We illustrate how having a tailor-made formalism for each aspect that is abstracted from other aspects is a key benefit of such a formal design framework. To clarify our discussions, we sketch an architecture specification and design method for reactive distributed real-time embedded systems.

In the approach we describe in this paper, we propose separating the concerns of computation, coordination, timing, and distribution, through different simple and abstract notations for these aspects. We also describe a weaving process that maps all these different aspects to a single semantic domain. The method is based on a formal semantics that should ultimately enable automated reasoning about designs. The idea exploited in this method can be extended to other aspects, and extended with more complex weaving criteria.

The remainder of this paper is organized as follows: Section 2 discusses computation, coordination, timing, and distribution as different aspects of a software design and suggests languages/notations to specify them. Section 3 contains a simple model of weaving the functional and non-functional aspects in a single semantic model. Section 4 proceeds with previous and related work, and Section 5 provides concluding remarks and research directions.

2 Exploring Aspects

This section focuses on the specification of computation and the three aspects coordination, timing, and distribution. We use a subset of GAMMA for specifying basic component functionalities (computations) and present its distinguishing features. We then present some ideas about specifying other aspects.

2.1 Modeling Computation with GAMMA

GAMMA is an abstract language, based on multiset rewriting on a shared data-space, designed to support parallel execution of a program on parallel and/or distributed architectures [3, 2]. The basic and atomic piece of functionality in GAMMA is the *rule*. The calculus of GAMMA [8] contains some composition operators to compose rules into programs. In [3], some patterns of rule composition (called *tropes*) are suggested to give hints for a program designer on how to compose/decompose functionalities to construct specific programs.

In this section, we focus on a subset of GAMMA involving the specification of basic rules. We thus factor out structuring decisions and make them a separate aspect model, namely the coordination model. The high level of abstraction proposed here follows from our basic assumption that a functionality specification model should only address essential computation for achieving the required computational functionalities. The other requirements which are non-functional, such as distribution, timing, etc., should be kept independent from functional requirements, as well as others so that change and evolution of it is localized and does not influence other parts of this or other aspects. The separation of basic functionality from coordination can also enhance reusability since a single functionality model may be reused with different coordination models to construct different programs or versions of a single program with different levels of efficiency [4]. For example, in expressing the requirements for an elevator control, the basic computation aspect includes the various signals and how they should behave with respect to a clock. The coordination aspect describes how the basic functionalities are composed to guarantee a correct functional behaviour. The distribution will localize some of the signals into different locations, and distribute the corresponding signal value computations. If signal values are shared across locations, the distribution aspect might express protocols for such shared accesses. The timing aspect, will describe constraints on signal behaviors.

Henceforth, the GAMMA model is only concerned with basic functionalities in the form of a simple input-computation-output pattern that abstracts from the following details:

1. *Relative ordering of actions (coordination)*. Basic functionalities (rules) are specified independently of each other. Hence, no special ordering of actions (control structure) is imposed on this particular specification.
2. *Timing*. The basic GAMMA model does not include any information about timing. Since it abstracts from ordering of actions, even a qualitative (causal) notion of time is not present in the GAMMA model.

3. *Distribution.* For any distributed system, the shared data-space is an abstraction that eases the programming, yet must be distributed in the implementation.
4. *Fault tolerance.* The GAMMA execution model requires programs to be designed in such a way that duplicated execution of atomic actions of a program cannot affect the functionality. Hence, replication of actions can be added transparently to the functional model.

The abstract nature of GAMMA in exploiting independent rewrite rules makes it suitable for definition of basic functionalities of software components. This does not mean that any of the above concerns are unimportant in system design and could be neglected completely. On the contrary, this abstractness provides the desired orthogonality, so that any of the above items can be specified and maintained as a separate aspect.

The syntax of a simple GAMMA program is given in Figure 1. A GAMMA

<i>Program</i>	$::=$ <i>ProgramName</i> = { <i>Rules</i> }
<i>Rules</i>	$::=$ <i>Rule</i> <i>Rule</i> , <i>Rules</i>
<i>Rule</i>	$::=$ <i>RuleName</i> = <i>MultisetExp</i> \mapsto <i>MultisetExp</i> \Leftarrow <i>Condition</i>
<i>MultisetExp</i>	$::=$ ϵ <i>BasicExp</i> <i>BasicExp</i> , <i>MultisetExp</i>
<i>BasicExp</i>	$::=$ <i>Variable</i> <i>Constant</i> (<i>Variable</i> , <i>BasicExp</i>) (<i>Constant</i> , <i>BasicExp</i>)

Fig. 1. Basic GAMMA Syntax

program consists of a non-empty set of rules, each rewriting the content of the shared multiset of data items. Execution of a program consists of applying rules to the multiset in arbitrary orders (sequential or parallel). Each rule consists of a set of terms valuated by multiset content values (this replacement is not necessarily unique for a specific rule and multiset). If a certain valuation of variables satisfies the condition in a rule, applying the rule results in removing the left-hand side valuations from the multiset and replacing them by the valuation of the right-hand side expression. We do not present the exact syntax of logical formulas in this paper – allowing them to be defined by selecting an appropriate underlying logic. However, we use the syntax and semantics of predicate logic formulas throughout this paper.

In [12], a formal operational semantics for GAMMA is given in the style of Plotkin [14]. Execution of a GAMMA program is based on execution of its individual rules. Hence, the main observable events in the semantics of such a program are changes in the shared multiset (multiset substitutions) and the termination of rules in case there are no enabling values to be found in the shared multiset.

The formal semantics defines the rules for executing single rules. It does not, however, suggest any order of rules to execute a program and therefore abstracts

away from the behavioral aspect of design. This suggests that execution of a naive GAMMA program allows any chaotic order on its rule executions.

Example 1. An elevator system, functionality aspect. Our elevator system consists of an elevator moving up and down between floors of a building (numbered from 0 to $MaxFloor$) to service requests. On each floor there is a push button to announce a request for an elevator when turned *on*. When an elevator arrives on a floor, the request flag is turned *off* automatically. The same setting works for the push buttons inside the elevator, which indicate the requested stops for passengers inside.

To model this distributed real-time system we propose a multiset containing events requesting an elevator stop represented by $((inStop, i), status)$ and $((extStop, i), status)$ that show the status of the request button for the i 'th floor, inside and outside the elevator, respectively. The tuple (cf, i) , shows where the elevator currently resides. The GAMMA program for the elevator system is given in Figure 2.

$$\begin{aligned}
ElevatorSystem = \{ & inRequest = ((inStop, i), off) \mapsto ((inStop, i), on), \\
& extRequest = ((extStop, i), off) \mapsto ((extStop, i), on), \\
& moveUp = (cf, i) \mapsto (cf, i + 1) \Leftarrow \\
& \quad \exists j; i < j \wedge ((extStop, j), off) \in State \wedge ((inStop, i), off) \in State, \\
& moveDown = (cf, i) \mapsto (cf, i - 1) \Leftarrow \\
& \quad \exists j; j < i \wedge ((extStop, j), off) \in State \wedge ((inStop, i), off) \in State, \\
& load = ((extStop, i), on) \mapsto ((extStop, i), off) \Leftarrow (cf, i) \in State, \\
& unload = ((inStop, i), on) \mapsto ((inStop, i), off) \Leftarrow (cf, i) \in State \\
& \}
\end{aligned}$$

Fig. 2. GAMMA Program for the Elevator System

The initial multiset for this system is defined as:

$$\begin{aligned}
State = [& ((inStop, 0), off), \dots, ((inStop, MaxFloor), off), \\
& ((extStop, 0), off), \dots, ((extStop, MaxFloor), off), \\
& (cf, 0) \\
&],
\end{aligned}$$

which shows that the elevator is at the ground floor initially and that there are no requests for the elevator.

2.2 Coordination

The functionalities described by GAMMA programs allow for many nonsensical executions of the system. In the elevator system, for example, the elevator can repeatedly move up and down between two floors without servicing any of the pending requests.

Regarding the model of separation of concerns proposed in this paper, we note that behaviour (both basic computations and coordination) is itself a complicated set of aspects that has been the main topic of discussion in the aspect-oriented community. As before, the abstractness of GAMMA is an elegant feature that allows us to define different composition, restriction, and ordering operators on basic rules.

To present our ideas about non-functional aspects, however, we present a simple and abstract model of basic functionalities in the GAMMA formalism, and do not discuss coordination in detail. The coordination of GAMMA programs is treated in [5] and for coordination of GAMMA programs including the timing aspect we refer to [12].

2.3 Timing

Timing constraints can be added to a specification to provide assertions regarding the execution time of GAMMA rules. This time is relative to the point from which the rule is selected for execution (when the previous rule execution is finished). We propose to add the timing aspect to a GAMMA specification by associating an interval to each rule name. This timing representation keeps the syntactic specification of timing separate from rule definitions, and hence allows independent change of both aspects. This method also allows a rule to have no timing assertion, which will be replaced by a default interval $([0, \infty])$ in the weaving process.

Since GAMMA rules assume a shared access to data, the timing aspect does not specify any assumptions about the cost of accessing the data items in a distributed setting. The above estimation is therefore only related to the computation time for each functionality. In the next section, we investigate the effects of putting constraints on the sharing/distribution policy.

Example 2. The elevator system, timing aspect. Suppose that the following timing information is given about the elevator system in Example 1:

- Pushing an internal or external button does not take time at all:

$$T_{inRequest} = T_{ExtRequest} = [0, 0].$$

- Going up and down between floors takes *StepTime* for each floor:

$$T_{moveUp} = T_{moveDown} = [StepTime, StepTime].$$

- The elevator will be loaded/unloaded within *MinService* and *MaxService* amount of time, depending on the number of people and goods waiting for it:

$$T_{load} = T_{unload} = [MinService, MaxService].$$

The timing information allows us to verify the timeliness of a functional specification, possibly for a given coordination, assuming the aspects are appropriately weaved together. In Section 3, we explain how this timing information can be weaved together with the functional specification into a single semantic framework.

2.4 Distribution

As expressed in Section 2.1, GAMMA abstracts from distribution of data and processing and assumes a shared multiset. Moreover, the timing aspect does not refer directly to the distribution model and accepts any distribution policy. Distribution is a major issue in complex systems, however, and should be taken into account and specified during software development. In this section, we study distribution as a separate concern.

To specify distribution, we need to specify the location of processes and data objects. Hence, we assume a set R containing rule names and a set T containing data types. Data types are used to categorize data items used/produced by different rules. We do not specify how to assign this typing to variables and constants but assume that there is a function from the sets of variables and constants to types ($type : Var \cup Con \rightarrow T$). The set of locations is denoted by P . Static distribution is defined as a function $StaticDist : R \cup T \rightarrow \mathcal{P}(P)$, representing the locations of the data objects and rules of each type. Note that we did not restrict locations to contain both data and processing (rules) and hence, a location may represent a storage node or processing unit, or both.

This general specification of distribution can be used to model more specific distribution policies, such as push and pull models. For example in a push model, the function $StaticDist$ should map any data type to its consumer side. In a pull model, however, the data type remains on the producer side and should be accessed (fetched) from the producer by the consumer.

We should note that preventing inconsistencies in accessing shared data items is still provided in the basic GAMMA semantics and need not be considered here. Nevertheless, if an application calls for its own dedicated consistency control algorithm, it should be specified in the form of stronger conditions in rules or an extension of the GAMMA model to a new aspect (by defining a notion of (in)dependence for parallel execution).

Example 3. The elevator system, distribution aspect. Suppose that sensors for request buttons on each floor are connected to the elevator via a fieldbus network. In this case, accessing the distributed locations will take some time from the elevator. To specify this model of distribution, we assume a location for the elevator and its internal buttons and a location for each external button. The distribution function for the elevator system then looks like the following:

$$\begin{aligned} StaticDist(type((extStop, i), status)) &= \{floor_i\} \\ StaticDist(type((inStop, i), status)) &= \{elevator\} \\ StaticDist(type(cf, i)) &= \{elevator\} \\ StaticDist(extRequest) &= \{floor_i \mid 0 \leq i \leq MaxFloor\} \end{aligned}$$

and for each rule $rule$ other than $extRequest$:

$$StaticDist(rule) = \{elevator\}$$

This distribution policy defines where the GAMMA rules $moveDown$ and $moveUp$ must look for remote copies of external request values from distributed locations.

This distribution model should be further combined with the functionality and timing model in one semantics in order to verify that the system satisfies properties that depend on the combination of several aspects.

3 Weaving Aspects

The idea of weaving is composing different aspects of design. In our case, we have to relate functionality, (coordination,) timing, and distribution, and present them in one semantic model. The orthogonality of non-functional aspects allows the designer of each aspect to neglect the other. As a result, the weaving process reflects change or even absence of one aspect in the whole semantics.

A GAMMA specification presents functionality in the form of independent rules. The timing specification aspect relates a rule to an interval representing duration of execution time. The distribution aspect defines the distribution of rules and data items over locations.

If there is no timing estimation specified for a rule (as it is the general case for un-timed specifications), it is assumed to be $[0, \infty]$, i.e., an arbitrary execution time. If the distribution aspect is absent, a single location is assumed. Our proposal for a formal semantics of weaving consists of a timed transition system [9] with transitions of a GAMMA program and timing consisting of computation time plus communication time.

We denote the computation time of a rule r by $comp(r)$. As mentioned before, if there is no interval defined for a rule r , $comp(r)$ results in $[0, \infty]$. This function induces a *by-name* weaving method to relate GAMMA rules and their respective timing estimations. In this paper, we assume that $comp(r)$ works as a function returning the execution time estimation of a rule, if available, or otherwise $[0, \infty]$. Nevertheless, this assumption could be relaxed by allowing several intervals associated to a rule, and hence letting $comp(r)$ return one of the intervals non-deterministically (or a set of intervals). This could be used to model the situation where a rule has multiple possible execution times, depending e.g. on varying implementation environments.

To represent communication costs resulting from the distribution policy, we use the function $comm(r)$, which returns the time cost for making local copies of the data items needed for the execution of rule r . For a rule r , $comm(r)$ is computed by taking the maximum of communication costs for all variables (of data items) v present in rule r , that reside in a different location than r . If all the data needed for the execution of a rule is available at the location of the rule itself, we assume the communication cost to be 0.

Example 4. Weaving of aspects of our elevator system. In Figure 3, a fragment of the timed transition system is given that results from weaving the computation, timing and distribution of the elevator system as described in previous examples. The transitions are labelled by the name of the rule(s) that are executed, the timing estimation of the execution, and the communication cost. For simplicity, only the relevant elements of the multiset contents are represented in this figure.

It is assumed that the time cost for communicating data from one node to another is CT .

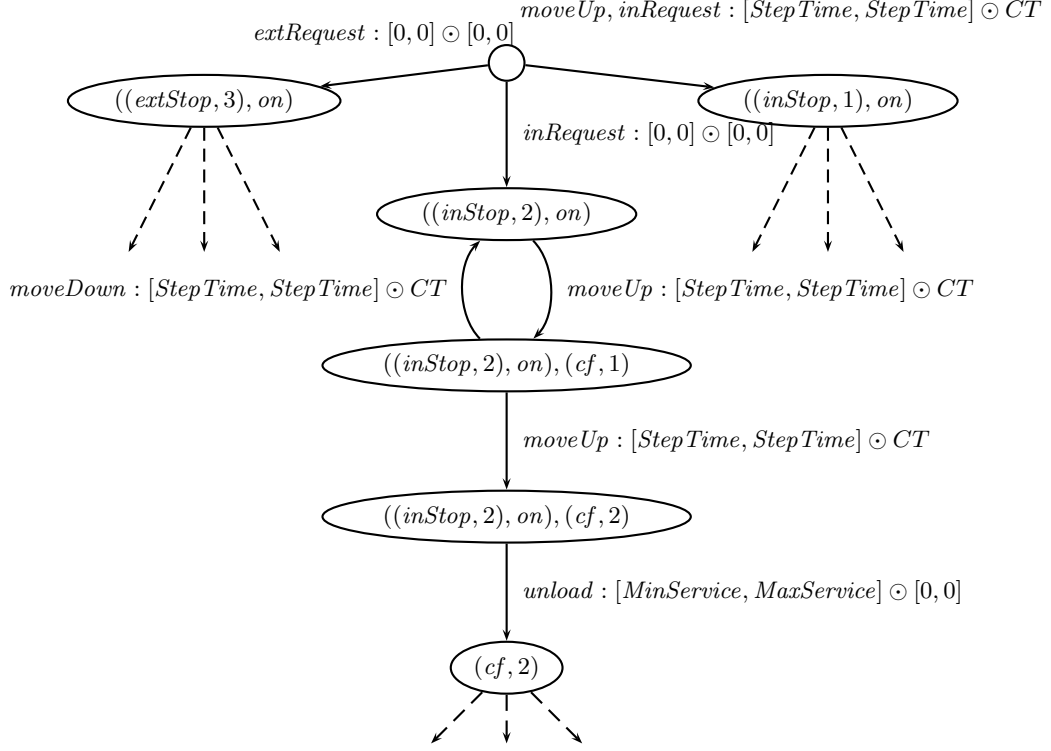


Fig. 3. Fragment of the Timed Transition System after Weaving.

The simple time weaving function presented here can be extended by adding estimations for failed attempts to execute a rule, or by defining the timing estimation as a function of multiset size or contents. In GAMMA, rule implementations, computation time and failure time may depend on the time for searching the multiset to find the appropriate valuation. These two extensions thus add to the practical value of the proposed method. Such extensions can illustrate the profit of the separation of concerns in the method outlined in this paper.

The timed transition system resulting from the weaving process allows the formal analysis and verification of the design. If the design satisfies the desired functional and non-functional properties, the aspect specifications can be used to (semi-)automatically generate an implementation through refinement and code generation.

4 Related Work

In [1], process algebra is suggested as a formal framework for aspect-oriented design. In the view of the author each aspect is described by a process term. By means of parallel composition with synchronization the aspects are combined. The elimination of the synchronization from the parallel composition of the aspects is considered the weaving of the aspects. Although there are many similarities (especially in the weaving) between the approaches, there are also some important differences. Firstly, the approach of Andrews does not enforce the description of basic functionalities separately from the coordination aspect, and secondly, we do not support the use of one and the same process algebra for the description of the different aspects.

In [11], an extension of the GAMMA formalism (namely Structured-GAMMA) is used as a specification language for an aspect-oriented implementation of a distributed shared memory protocol. There, the authors mention the benefits of abstraction from distribution in the GAMMA programs and the possibility of formal reasoning using this specification language.

5 Conclusion and Future Research

The current trends in AOP [6] can be summarized as follows:

1. Semantic correctness of aspects and compositions.
2. Defining methods for identifying and specifying canonical models for cross-cutting concerns, including methods for composing aspect models.
3. Defining formal models for determining functional and quality characteristics of crosscutting concerns individually and together.

We can summarize our contribution to these challenges as follows:

1. We provided some ideas for the formal design of a small number of aspects, mainly related to distributed real-time systems, which are kept separate and abstract from each other.
2. These aspects are weaved together into a single semantic framework.

The main challenges in our future research are the following:

- Providing a formal syntax, weaving, and semantics of the aspects discussed in this paper. In [12], a formal syntax and semantics are given for basic functionality, coordination, and timing aspects.
- Extension of the method sketched in this paper to other aspects such as power-awareness, fault-tolerance, persistency, etc.
- Developing/studying logics for expressing properties of the aspect models and the weavings of those.
- Performing case studies to validate the method.
- Developing automated design methods and tools that support the aspect weaving process the reasoning in the aspect models, and the refinement towards implementation.

References

1. J. H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, volume 2192 of *Lecture Notes in Computer Science*, pages 187–209, Berlin, 2001. Springer-Verlag.
2. J.-P. Banâtre, P. Fradet, and D. Le Métayer. Gamma and the chemical reaction model: Fifteen years after. In C. S. Calude, G. Paun, G. Rozenberg, and A. Salomaa, editors, *Multiset Processing: Mathematical, Computer Science, and Molecular Computing Points of View*, volume 2235 of *Lecture Notes in Computer Science*, pages 17–44. Springer-Verlag, Berlin, 2001.
3. J.-P. Banâtre and D. Le Métayer. Programming by multiset transformation. *Communications of the ACM (CACM)*, 36(1):98–111, Jan. 1993.
4. M. R. V. Chaudron. Separation of Correctness and Complexity in Algorithm Design. Technical Report 94-36, Leiden, The Netherlands, 1994.
5. M. R. V. Chaudron. *Separating Computation and Coordination in the Design of Parallel and Distributed Programs*. PhD thesis, Department of Computer Science, Rijksuniversiteit Leiden, Leiden, The Netherlands, 1998.
6. T. Elrad, R. E. Filman, and A. Bader. Special issue on aspect oriented programming. In *Communications of the ACM (CACM)*. ACM Press, 2001.
7. G. Smith. *The Object-Z Specification Language*, volume 1 of *Advances in Formal Methods*. Kluwer Academic Publishers, Boston, 2000.
8. C. L. Hankin, D. Le Métayer, and D. Sands. A calculus of Gamma programs. In *Proceedings of the Fifth International Workshop on Languages and Compilers for Parallel Machines*, volume 757 of *Lecture Notes in Computer Science*, pages 342–355, Berlin, 1993. Springer-Verlag.
9. T. A. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In J. W. de Bakker, K. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Proceedings REX Workshop on Real-Time: Theory in Practice*, Mook, The Netherlands, June 1991, volume 600 of *Lecture Notes in Computer Science*, pages 226–251, Berlin, 1992. Springer-Verlag.
10. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, New York, NY, June 1997. Springer-Verlag.
11. D. Mentré, D. Le Métayer, and T. Priol. Towards designing SVM coherence protocols using high-level specifications and aspect-oriented translations. Proceedings of the 1st Workshop on Software Distributed Shared Memory, Rhodes, Greece, June 1999.
12. M. Mousavi, T. Basten, M. Reniers, M. Chaudron, and G. Russello. Separating functionality, behaviour and time in the design of reactive systems: (GAMMA + coordination) + time. To appear, 2002.
13. P. Tarr and H. Ossher and W. Harrison and S.M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, pages 107–119. ACM, May 1999.
14. G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, Sept. 1981.
15. AspectJ Website. <http://www.aspectj.org>.

Aspect Oriented Software Architecture: a Structural Perspective¹

A. Navasa, M.A. Pérez, J.M. Murillo, J. Hernández

Computer Science Department. University of Extremadura
Escuela Politécnica. Avda. de la Universidad s/n 10071 Cáceres. Spain
{amparonm, toledano, juanmamu, juanher}@unex.es

Abstract. The positive results obtained by researchers on aspect-oriented programming during the last few years are promoting the aim to export their ideas to the whole software development process. One of the stages in which Aspect Oriented Software Development (AOSD) techniques can be introduced is software architectural design. This would make design of complex systems an easier task whilst cost development, cost maintenance, reuse, etc., would be improved. However, integrating both architectural design and aspect orientation is a non-trivial task. The different nature of the aspects that can be involved in an application and the different requirements that applications impose on the treatment of aspects make it difficult to handle aspects in a uniform way while at the same time preserving the simplicity of the design process. In this study the structure of the problem of separation of crosscutting concerns in the architectural design based on the aspect oriented approach is analysed. The analysis will identify the kind of changes that must be made in the current technology to manage this sort of integration. In particular, this paper focuses on how aspects separation can be handled by Architecture Description Languages and architectural styles. The aim of this article is not to propose a particular solution but to propose some general guidelines on which solutions can be based.

1. Introduction

During the last few years, a great amount of work has been done to propose AOP techniques [Kic+96]. Time to harvest results has arrived and the results have shown the real benefits of using aspect orientation in the application development process. These include: increase in productivity, re-usability and adaptability. Thus, Aspect Orientation has been shown to be so relevant that there is an increasing demand for it to be extended to the whole software development process [Cla+99, Nak99, ChLu01]. So, concepts from AOP are being extended to early stages in the software lifecycle, generating new disciplines like Aspect Oriented Software Development (AOSD) and Aspect Oriented Design (AOD).

One of the stages on which AOD is focused is the architectural design of applications [Gru00]. It seems reasonable to observe software architecture from the aspect-oriented point of view due to several reasons. On the one hand, there are aspects that are inherent to the systems themselves and, consequently, they could be treated in their architectural design. On the other hand, aspects separation in the systems architecture would make design an easier task, improving cost development, making it easy to reuse designs, reducing maintenance cost and so on. However, aspect orientation and architectural design are disciplines that have evolved separately and their integration is not as trivial as it may seem.

In this paper some methodological considerations about how to face the integration of the aspects separation in the architectural design are presented. Thus, this work examines the separation of crosscutting concerns during software architecture design from a structural point

¹ This work has been supported by CICYT, project TIC 99-1083-C2-02

of view instead of a morphological one by studying the structure of the problem and extracting general conclusions more than proposing a particular solution. This reasoning will conclude that aspects separation at architectural level have some similarities with typical coordination problems solved using coordination models and languages [Car92]. Leaning on this, the kind of changes that must be introduced in current Architecture Description Languages to manage the aspect separation when designing software systems is discussed.

In section 2, the difficulties that appear when integrating aspects separation in the current state of software architecture will be shown. Next, in section 3, the process of how an application can be designed by handling separately the non-functional aspects intervening on it will be analysed. This analysis will show how separations of crosscutting concerns at architectural design can be handled as a coordination problem. Finally, section 4 shows the conclusions and outlines future works.

2. Software architecture and Aspect Orientation

During the last few years, software architecture design techniques have been acquiring more and more relevance as the complexity of systems has been growing. The experience has demonstrated the benefits of using these techniques when designing such systems. Software architecture allows the software designer to specify the systems structure in terms of components and connectors. Components specify the functionality of the system whilst connectors determine the interaction between components. In these terms, software architects can concentrate on the structural properties of the systems avoiding the implementation details. This makes possible to face complex systems reducing cost and developing time.

Two of the most useful tools for software architects are both architectural styles and Architectural Description Languages (ADL). Architectural styles give rules to build system families with similar characteristics. A number of architectural styles that direct the work of the software architects has been catalogued [ShGa96]. ADL are languages that provide primitives to specify components and connectors. Several ADL have been developed with different features: Rapide [Luc95], Darwin [MaKr96], Wrih [All97], Acme [GaMoWi97], etc..

In order to introduce the concepts from aspect orientation in the architectural design of applications, it would be necessary to adapt the current software architect s tools. However, some previous works of our research group [NaPeMu01, PeNaMu01, NaPeMu02] has revealed that this is a non-trivial task. With respect to the architectural styles, one may consider proposing an architectural style for systems in which aspects will be handled in a separated way. Nonetheless, our experience tells us that it is not easy for the following two reasons:

1. On the one hand, the different nature of the aspects intervening in a system makes it difficult to handle them in a simple and uniform way using a single style.
2. On the other hand, the same aspects in different systems could require different treatment and this makes it difficult to solve the problem in a simple way. For example, an application with real time and distribution constraints may require treating the real time constraints before, and then the distribution constraints. A second application with the same real time and distribution constraints may require treating the aspects in the opposite order. Therefore, it would be difficult again to use a single architectural style for the two applications.

With respect to ADL, current languages as the ones mentioned before do not provide primitives to specify the aspects separation. ADL are designed to specify components and the connections between their interfaces. Now, it would be necessary to specify connections not only between components but also between components and aspects and aspects with others aspects. However the idea of having interfaces on the aspects is not clear. It is not even clear how to specify aspects.

All the above allows us to conclude that new mechanisms are required to introduce aspects separation during architectural design. The next question is, what must be the nature of such a mechanism?. To answer this question a structural point of view was adopted.

3. Aspect Separation at the Architectural Level: a Coordination Problem

In this section it is analyzed how separation of crosscutting concerns can be handled in architectural design of software systems. The analysis will allow us to extract some general conclusions. The structure of the faced problem is depicted in figures 1a and 1b

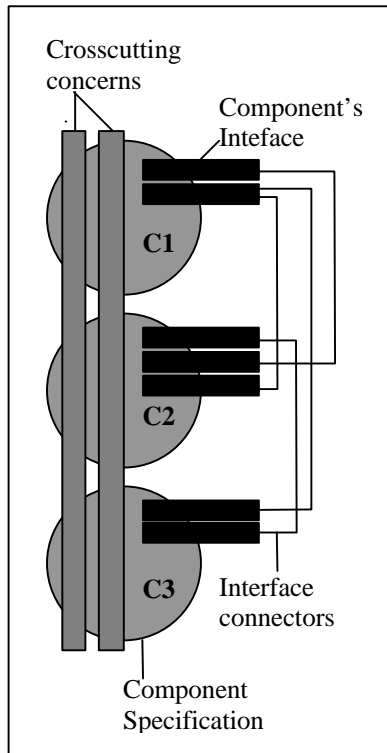


Fig 1a. Current specification of software architecture

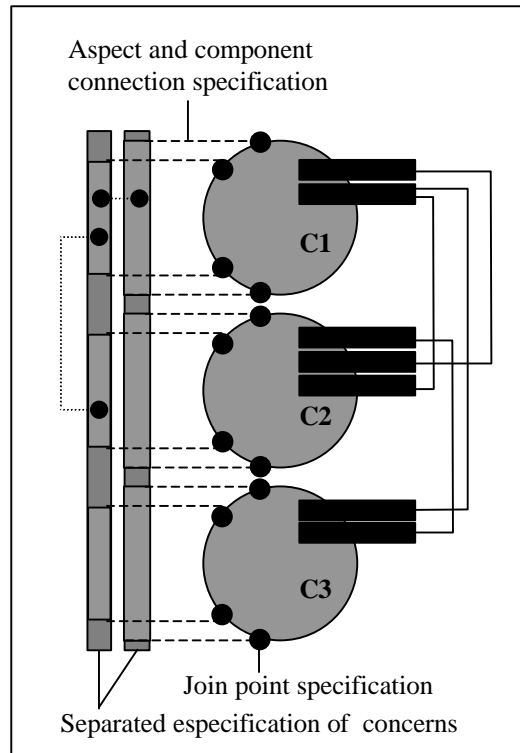


Fig 1b. Separated specification of crosscutting concerns at the software architecture level

Let us suppose that figures 1a and 1b specify the software architecture of the same system. Figure 1a shows how software architectures are currently specified. The software architects have to specify the functionality of the system using components and the interaction between components using connectors. However the specification of the concerns intervening in the system is crosscutting the specification of components. On the other hand, introducing concepts from aspect orientation in architectural design must lead to a situation as the one described in figure 1b.

In figure 1b the specification of aspects has been separated. Now the software architect has to specify components with their interfaces, connections between interfaces and the separated aspects. However, in order to produce an equivalent design to the one showed in figure 1a software architects must carry out the following tasks:

1. To specify in components the points from which the specification of the aspects has been extracted. That is, the software architect have to specify the **join points** in the specification of components.
2. To specify the connections between aspects and join points. Such connections have a different nature that connection between interfaces. The purpose of such connectors is to sort

out the specifications of the whole system in order to maintain the coherence of the original system preserving its semantic. So, such connectors have to specify when and how each aspect must be treated. This is a typical problem of coordination that has been solved with coordination models and languages [Gpap98].

Having all the above in mind it can be concluded that to manage the separation of concerns at architectural design of software systems the current ADL must be extended in order to provide the following functionality:

1. To specify functional components with interfaces and connection between interfaces. This functionality is already provided by current ADL. However, new ADL should make possible to specify join points in functional components. So new primitives must be provided in order to manage the specification of the joint points. These primitives should support the specification of every kinds of joint points not restricted to a predefined set of them.
2. To specify aspects. Aspects have a different morphology that functional components in the sense that they do not provide services and interfaces. However, in order to modularize aspects, they would be specified as a special kind of component described by new primitives.
3. Finally, to specify connectors between join points and aspects. As it has been introduced before the purpose of such connectors is to sort out the specification of the whole system. This problem is already solved by coordination models and languages. So it must be studied the possibility to import solutions from this area. In particular, exogenous coordination models [Arb98, Frø96, Mur99 to name a few] specify the code to determine how functional components coordinate in separated entities from those to be coordinated. Different approaches support different kinds of functional and coordination components. However, in all proposals the coordination components sort out the global execution of the tasks implemented by the system. The same schema could be used here to specify connectors between aspects and components. In this case there are two different kinds of components: functional and aspect components, and connectors could be coordination components. The mission of such components would be to specify when and how the aspect components must be treated.

The sort of aspect components and connectors mentioned before could be reused from one system specification to another. Patterns of usual aspects and connectors could be deduced and, in this way, aspect-oriented architecture styles could be proposed.

4. Conclusions and future works

In this paper some guidelines to integrate concepts from aspect orientation in architectural design of software systems have been presented. In particular the work is focused on the way in which aspect separation can be handled by means of architectural styles and Architecture Description Languages. Based on the previous experience of our research group it is concluded that it is very difficult to deal with such integration preserving the simplicity during the design process. Then the structure of the problem of separation of concerns at the architectural level has been analysed. This analysis has allowed us to extract two important conclusions. First, aspect separation at the architectural level can be reduced to a coordination problem. Second, based on the above, a proposal has been put forward regarding the kind of changes that must be made in the current tools for architectural design in order to support the separation of concerns.

Currently we are working along different lines. In particular, in order to be able to specify the places where aspects must be treated, the way in which joint points can be characterised in components is being analysed. Also, to specify the order and the treatment that aspects must receive, we are studying the nature of the connectors. To specify this sort of connectors we are studying the possibility of using **Rew** [Arb02] from Farhad Arbab. Farhad Arbab has a recognised experience as a researcher in the area of coordination models and languages. He is

the father of the exogenous coordination model IWIM. **Rew** is a channel-based exogenous coordination model wherein complex coordinators, called "connectors" are compositionally built out of simpler ones. These features make connectors a simple and powerful mechanism suitable to be used here. Finally, work is being done to determine the way in which the specification of aspects can be reused from one system's architecture to another. In particular it is being studied how the trading of aspect components could be done.

Acknowledgements

We would like to thank the anonymous referees for their helpful comments.

References

- [AlChNo01] J. Aldrich, C. Chambers, D. Notkin. ArchJava: Connecting Software to Implementation. Workshop on Language Mechanism for Software Component in OOPSLA 2001 Octubre 2001 Tampa Bay, Florida, USA
- [All97] R. Allen. *A Formal Approach to Software Architecture*. Tesis Doctoral. School of Computer Science. Carnegie Mellon University, USA CMU-CS-97-144. 1997.
- [AlDoGa98] R. Allen, R. Douence, D. Garlan. CMU / JRJSA-JNRJA /CMU. Proceeding of 1998 Conference on Fundamental Approaches to Software Engineering. Lisbon, Portugal Marzo 1998
- [AkTe98] M. Aksit, B. Tekinerdogan. *Solving the Modeling Problems of Object-Oriented Languages by Composing Multiple Aspects Using Composition Filters*. Workshop AOP. Bruselas. 1998.
- [Arb98] Farhad Arbab. What Do You Mean Coordination? Bulletin of the Dutch Association for Theoretical Computer Science (NVTI). March 1998.
- [Arb02] Farhad Arbab and Farhad Mavaddat. *Coordination Through Channel Composition*. F. Arbab and C. Talcott (Eds.). 5th International Conference, COORDINATION 2002, YORK, UK. LNCS 2315, Springer-Verlag. April 8-11, 2002.
- [Car92] N. Carreiro, D. Gelernter. Coordination languages and their significance. *Communications of the ACM*, 35 (2):97-107, February 1992
- [ChLu01] Christina von Flach G. Chávez and Carlos J. P. De Lucena. Design-level Support for Aspect Oriented Software Development. Doctoral Symposium OOPSLA 2001 Octubre 2001 Tampa Bay, Florida, USA
- [Cla+99] Siobhán Clarke, William Harrison, Harold Ossher, Peri Tarr. Separating Concerns Throughout the Development Lifecycle. ECOOP'99. Workshop on Aspect-Oriented Programming. Lisbon. Portugal. 1999.
- [Frø96] S. Frølund. Coordinating Distributed Objects. An Actor-Based Approach to Synchronization. *The MIT Press*. 1996.
- [GaMoWi97] D. Garlan, R.T. Monroe, D. Wie. *Acme: An Architecture Description Interchange Language*. In proceeding of CASCON'97, Ontario. Canada. 1997.
- [Gpap98] G.A. Papadopoulos, F. Arbab. Coordination Models and Languages. *Advances in Computers*, 48. Academic-Press, 1998.
- [Gru00] J. Grundy. A Method and Support Environment for Distributed Software Component Engineering. Proceeding of 2000 Conference on Software- Methods and Tools. Wollongong, Australia 2000
- [Ki+96] G. Kiczales et al. *Aspect-Oriented Programming*. In Max Mühlhäuser ed, Special Issues in Object-Oriented Programming, Workshop Reader of the 10th. European Conference on Object-Oriented Programming, ECOOP 96, Dpunkt-Verlag. 1997.
- [Ki+01] G. Kiczales, E. Hilsdale, J. Hugunim, M. Kersten, J. Palm, W. G. Griswold. Getting Started with AspectJ. *Communications of the ACM*, October 2001, vol 44, num 10.
- [LiLoMi00] K. Lieberherr, D. Lorenz, M. Mezini. *Programming with Aspectual Component*. Northeastern University, EEUU. 2000.
- [Luc95] D.C. Luckman et al. *Specification and Analysis of Systems Architecture using Rapide*. IEEE Transactions on software Engineering. San Francisco. EEUU. 1995.

- [MaKr96] J. Magee, J. Kramer. *Dynamic Structure in Software Architectures*, en ACM Foundations of Software Engineering. San Francisco. EEUU. 1996.
- [Mur99] J.M. Murillo, J. Hernández, F. Sánchez, L.A. Álvarez. Coordinated Roles: Promoting Reusability of Coordinated Active Objects Using Events Notification Protocols. P. Ciancarini and A. L. Wolf (Eds.). *Third International Conference COORDINATION 99*. Amsterdam, The Netherlands. Springer-Verlag, LNCS 1594. April 1999.
- [Nak99] Shin Nakajima. Separation of Concerns in Early Stage of Framework Development. Workshop on Multidimensional Separation of Concerns. OOPSLA 2001 Octubre 2001 Tampa Bay, Florida, USA
- [NaPeMu01] A. Navasa, M.A. Perez, J.M. Murillo, Developing Components Based Systems using AOP Concepts. Workshop on Advanced Separation of Concerns in ECOOP 2001. Budapest Hungría
- [NaPeMu02] A. Navasa, M. A. Pérez, J.M. Murillo Definición de un estilo arquitectónico para desarrollos software de sistemas complejos, basado en separación de aspectos. Cáceres, Informe Técnico de la Universidad TR-13/2002. Febrero 2002.
- [OsTa01] H. Ossher, P. Tarr. Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software. Communications of the ACM, October 2001, vol 44, num 10.
- [PeNaMu01] M.A. Perez, A. Navasa, J.M. Murillo. An Architectural Style to Integrate Components and Aspects. Workshop on Feature Interaction in Complex Systems in ECOOP 2001. Budapest Hungary
- [ShGa96] M. Shaw, D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall. 1996.

Towards an Aspect-Oriented Approach to Improve the Reusability of Software Process Models

Rodrigo Quites Reis^{1,3} Carla Alessandra Lima Reis^{1,3} Heribert Schlebbe² Daltro José Nunes³

¹ Departamento de Informática, Universidade Federal do Pará (UFPA), Belém, PA, Brazil

² Fakultät Informatik, Universität Stuttgart (Uni-Stuttgart), Stuttgart, BW, Germany

³ Instituto de Informática, Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, RS, Brazil

URL: <http://www.inf.ufrgs.br/~prosoft>

E-mail: quites@computer.org, clima@ufpa.br, schlebbe@informatik.uni-stuttgart.de, daltro@inf.ufrgs.br

Abstract

The Aspect-Oriented (AO) paradigm is a recent trend to assist the development of high complexity software, proposing specific language level constructs to separate different concerns of software. This paper investigates the adoption of AO concepts to assist the design of high-level management policies in software process models. Process Policies are proposed in this text as first order modeling constructs to express syntactical and instantiation aspects of process models, complementing the role played by standard Process Modeling Languages (PMLs). The proposed approach is a consequence on the recent evolution in PML design to meet the recent advances in the Software Architecture field. Finally, the proposed model is briefly discussed with respect to its performance on describing standard software process examples, pointing to some required future improvements.

Keywords: Software Process Technology, Aspect-Oriented Software Development, Architecture Description Languages, Process Modeling Languages.

1 INTRODUCTION

Software is the base for a number of organizations involved in different activities around the planet, consisting on a strategic element to differentiate current products and services. Nowadays, software is embedded in a number of systems related to a wide selection of sciences and technologies, and according to Pressman [PRE01], software “will become the driver for new advances in everything from elementary education to genetic engineering”.

In spite of recent advances on Software Engineering, software community still discusses about the low quality and productivity of software industry, which have consequence on users’ satisfaction. In addition, requirements for new software applications increase in size and complexity, while managers expect to short the time to market. Thus, current software development processes are complex, involving a high quantity of specialized personnel that must work cooperatively among different locations, in an environment with technical and cultural challenges.

Software Process Technology (SPT) has been proposed to manage complex software development [FEI93]. Thus, tools, languages and methodologies are presented by the literature to improve industry-adopted software process models. Nowadays, SPT is profoundly influenced by the languages used during software process modeling, which are generically known as Process Modeling Languages (PML) [FEI99]. PMLs are used to formally describe software development processes through the precise specification of process tasks, components, and people influence on process results.

This text presents an investigation on the feasibility of providing separation of concerns to support the design of reusable process models. In addition, it analyzes the adoption of some ideas from the Aspect-Oriented (AO) paradigm in order to complement standard PML paradigms. The investigated hypotheses are:

- The concepts of Aspect and Crosscutting concerns are not tied to specific software paradigms (e.g., the Object-Oriented paradigm) or even to the software development domain.
- The complex, multiple viewpoint reality captured by software process models would greatly benefit from AO Programming concepts.

Therefore, this paper evaluates the feasibility on using policies as first order crosscutting properties for an activity-oriented PML called *APSEE*. The paper is organized as follows. Section 2 presents an overview on Software Process Technology. Section 3 briefly presents the underlying meta-model used for process modeling. Section 4 introduces the *StaticPolicy* and *InstantiationPolicy* languages. Section 6 discusses related work and presents the final remarks.

2 SOFTWARE PROCESS TECHNOLOGY

Competitive pressures on software industry have encouraged organizations to examine the effectiveness of their software development and evolution processes. According to [PAU94], a typical goal of an organization focused on achieving higher capability levels is to document software process and define one or more ideal processes to strive for.

Process-Centered Software Engineering Environments (PSEEs) constitute a special kind of Software Engineering Environments (SEEs) that support the rigorous definition of software processes, aiming to analyze, simulate, enact and reuse processes definition. Research groups and software industry developed many PSEEs during the last decade, including EPOS, ADELE, SPADE, PEACE+ and E³ (cited in [DER99]).

Complex software development involves activities performed by a number of developers with different technical

skills. Automated control is possible using a process model: many kinds of information are integrated in this model in order to identify “who, when, how and why the steps are performed during software development” [LON93]. Therefore, Process Modeling Languages (PMLs) are available to facilitate the description and manipulation of process models.

Software Process Reuse currently defines a wide area of research and practice related to different aspects of the reuse of knowledge obtained from previous successful projects. The specialized literature describes a number of descriptions of generic process models that can be reused in different contexts, varying from textual (narrative) descriptions to semi-formal process models. While informal process models are useful to describe methodologies for software development, the lack of rigor on the description of software process models inhibits their automation. Informal process models are also consequence of the lack of success on promoting adequate support for process modeling.

The current practice on describing reusable process models often relies on process designers' knowledge. Since the provocative proposal of Osterweil to evaluate the effectiveness of using standard software formalisms to describe software processes (“Software Processes are Software Too” [OST87]), the research on process modeling has been focused on adapting the existing solutions from (standard) product-oriented technology. In fact, recent studies highlighted the similarity of Architecture Description Languages (ADL) and Process formalisms [EST99]: both intend to offer high level language constructs (i.e. capable of providing a global picture of the process) that are executable (i.e. the communication between components can be automated).

Furthermore, while at a first sight the challenge of describing reusable process elements appears to be equivalent to the traditional software reuse problem, this is only partially true, since a process model mixes social, organizational, technological and environmental issues. The increasing complexity on software process modeling implies in the investigation on using successful general-purpose software development technologies in the process modeling field, as described next.

3 THE APSEE MODEL

Since the topic under discussion by this paper is a part of a larger work, where process technology is investigated in order to increase the level of automation for process management, this section presents the main characteristics of the proposed model to store reusable process descriptions and its underlying infrastructure.

3.1 The underlying APSEE meta-model

While a complete description and evaluation of the APSEE meta-model is out of the scope of this paper, this section presents an overview of the underlying infrastructure w.r.t. its historical roots and design characteristics.

APSEE is a software framework to automate software process management that evolved from a Software Process Engine [LIM98] originally proposed for the PROSOFT environment. PROSOFT is a formal object-based software development paradigm [NUN92] that is currently implemented as a Java-based SEE [SCH97] (in cooperation program between PPGC-UFRGS - Porto Alegre, Brazil - and Uni-Stuttgart - Germany). Nowadays, APSEE is the underlying integration kernel for a number of meta-models to support process simulation, instantiation, enactment and improvement and reuse [REI02a]. Figure 1 presents a couple of screenshots describing different views of an enacting process.

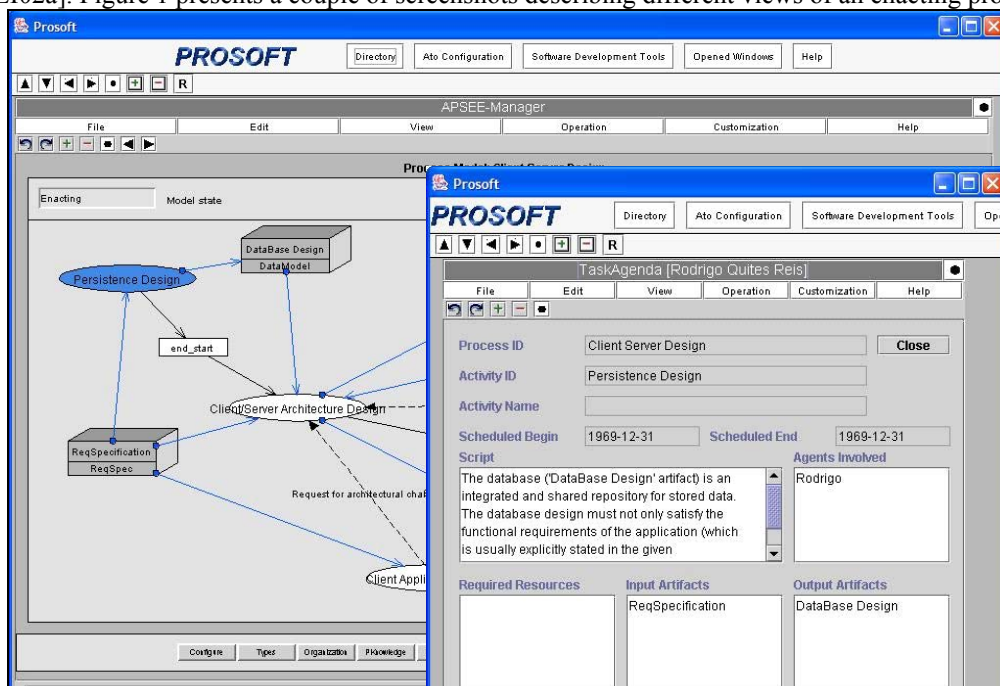


Figure 1 Snapshots of Manager (background – process monitor) and Developer (foreground - agenda) views of process enactment under the APSEE system

The *APSEE* meta-model is based on an activity-based paradigm, describing a process as a partially ordered collection of activities. A graph-based activity-oriented graphical PML is available (*APSEE-PML*), providing graphical representation for the proposed set of language constructs. A dynamic set of control mechanisms is available, delivering flexible synchronization on activity connections.

Three main process types are available, representing its engagement degree with details of instantiation of the model to its context. A **Process Template** supplies generic descriptions for posterior reuse. **Instantiated Processes** have information about its allocation. Finally, **Enacted Processes** store information about actual processes performance.

3.2 The *APSEE* System Architecture: separation of concerns for process models

The design of the *APSEE* architecture was profoundly influenced by the need to provide support for explicit separation of concerns during process modeling. According to [PER96], during the development of a process model its designer deals with a multi-dimensional reality that mixes information about the **Project** (i.e., instantiation details like organization-specific roles and detailed schedule), the **Environment** (the set of software tools needed to perform a development step/activity) and the **Process** itself. The proposed model extends the ideas of Perry by defining additional dimensions: **People** (including details about role types, agents, groups and abilities), **Software** (describing produced, consumed and transformed software artifacts) and **Resource** (describing consumable, exclusive and shareable resources).

The proposed *APSEE* Architecture is informally depicted as a set of layers in figure 2¹. The Processes layer is the most important since it defines processes and activities that are associated to components belonging to different layers during modeling and enactment time. Each layer describes elements that are defined independently (and are bound together during the construction of process models). It is important to note that the current system offers graphical representation in the PML notation only for Processes and Software components.

Figure 2 shows the connections among the enabled Policy instances. The solid lines describe to which process component the Policy instance is enabled. In the example, “A” and “B” constitute typical static policy instances (since they do not require additional information), while “C” and “D” instances constitute Resource Instantiation examples (since they are also associated to specific resource types). Finally, “E” is an example of an Agent Instantiation Policy: for a specific process activity it will apply some user-defined criteria to assist the selection of ‘Programmer’ agents.

The provided composition protocol is formally defined using a combination of algebraic methods (Graph Grammars and Algebraic PROSOFT [NUN92]) and it is further described in [REI02a].

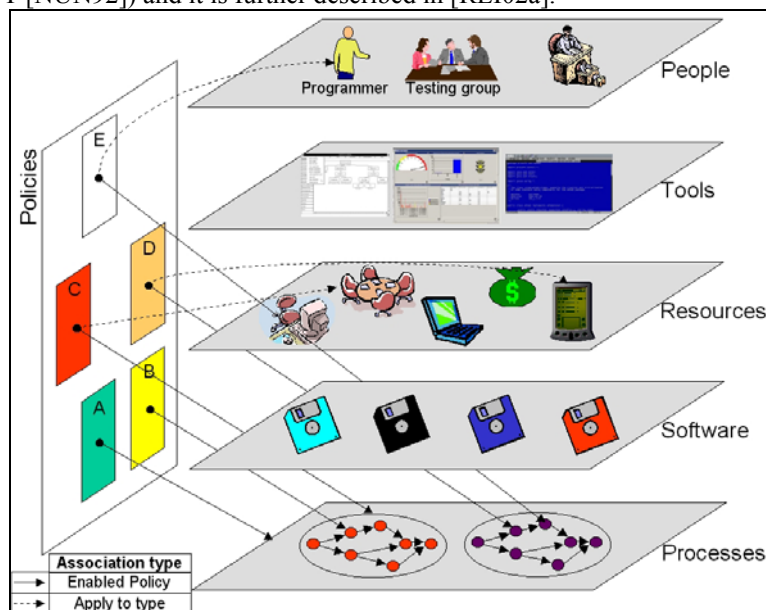


Figure 2 Orthogonality of Policies and Process Definitions

4 USING POLICIES AS ORTHOGONAL ELEMENTS FOR PROCESS MODELING

Process policies constitute “the guiding principles for process development and/or enactment” [FEI93]. In the *APSEE* model, the Policy constructor was designed to solve the need to store information about generic management strategies that would be difficult to express using standard (i.e., activity, rule or activity-oriented) PML constructs. Policies complement *APSEE-PML* by describing generic management strategies across different process components. Thus, this section highlights the influences from AO Paradigm to the design of the proposed Policies constructs.

4.1 The Aspect-Oriented Paradigm for Software Development

The AO paradigm has been proposed as a technique to improve the separation of concerns in software design and

¹ The Projects (i.e., Enacted Processes) layer is intentionally not presented in the layered diagram of figure 3, neither the relationship among the Tools, People, Resources and Software layers.

implementation. The main idea is that while the standard hierarchical modularity mechanisms provided by the OO paradigm are useful, they are inherently unable to modularize all concerns (properties) of interest in complex systems. Aspect-oriented software engineering provides mechanisms that explicitly capture the crosscutting system structure. Thus, the goal of AO software design is to support the developer in cleanly separating components (objects) and aspects (concerns) from each other, by providing mechanisms that make it possible to **abstract** and **compose** them to produce the overall system. Aspects are defined as properties that crosscut components in system's design.

Separating concerns from components requires a mechanism for composing – or *weaving* – them later. Central to the process of composing aspects and components is the concept of *join points*, the elements of the component language semantics handled by aspect specifications. Join points are well-defined points in the dynamic execution of the program. Examples of join points are method calls and receptions, method executions, and field sets and reads.

4.2 Process Policies

Figure 3 presents the correspondence of *APSEE* and AO constructs, w.r.t. the proposed policy modeling paradigm. The fundamental ideas that guided the recent evolution of AO and separation of concerns influenced the design of the *APSEE* Policy construct. The proposed approach for Policy definition, composition and enactment are similar to the ideas of Kenens [KEN98]: aspects (policies, in the adopted terminology) are available to handle static (syntax) and dynamic (instantiation) elements of the specific domain of process models.

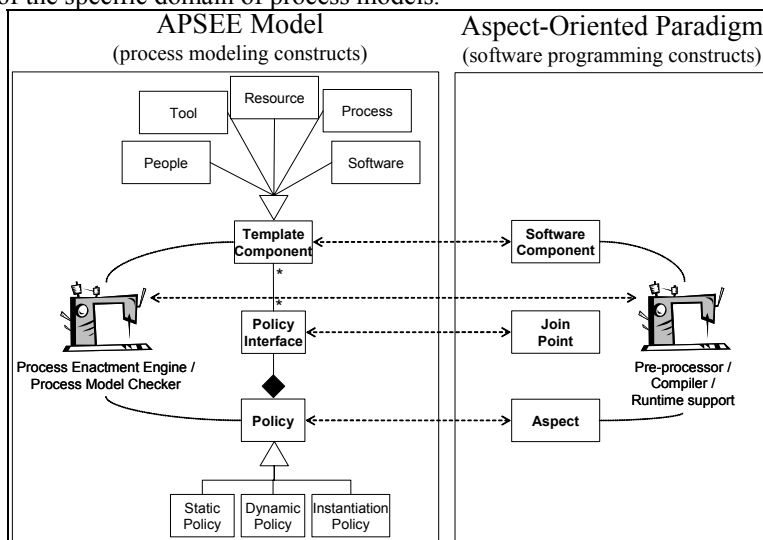


Figure 3 Mapping of concepts between the *APSEE* Model and the Aspect-Oriented Paradigm

Policies are active pluggable entities that restrict the syntactical formation of process models in response to static or dynamic process properties. Each Policy instance defines an Interface connector to the existing *APSEE* component types, establishing the required join points for the proposed approach. During enactment, the execution deviates from the normal process enactment flow to execute the associated policies, which are written with specialized languages. Three Policy types are available in the *APSEE* model: (Resource and People) Instantiation, Dynamic and Static Policies. The UML class diagram of figure 4 describes the main classes involved with Policies' definition. For the sake of brevity and clarity, this text presents next only the Static Policy and Resource Instantiation types.

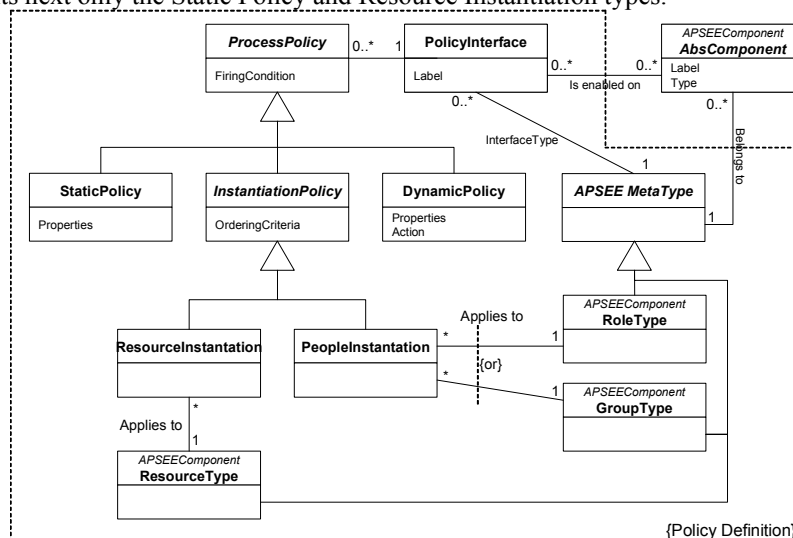


Figure 4 The Policies' data types

4.2.1 Static Policies

From a syntactic point of view, StaticPolicy main components are: its **name**, the **policy interface** (specifying the type to be observed), the **enabling properties** (logical pre-conditions to enable the policy – the *FiringCondition* attribute in figure 4), and the **properties** (an ordered list of logical conditions to be checked by *APSEE* before releasing a process model to enactment). Figure 5 shows a small example: it ensures that a Development activity must be followed by a Verification activity (involving different sets of agents), which, in turn, must have access to all artifacts manipulated by the development activity. The complete description of StaticPolicies semantics and additional examples are in [REI02a].

External Review: Static Policy Details	
Policy Interface	
Label	Type
a	<input checked="" type="radio"/> Activity <input type="radio"/> Resource <input type="radio"/> Agent
Enabling Properties	
a.get_type() sub_type_of "Development"	
Properties	
a.number_of_successors() = 1 and a.get_successors().all.get_type() sub_type_of "Verification" and a.get_successors().all.get_agents() not_contains a.get_agents() and a.get_successors().all.get_input_artifacts() contains a.get_output_artifacts() union a.get_input_artifacts() and a.get_successors().all.get_output_artifacts().any.get_type() sub_type_of "Review Report" and a.get_successors().any.is_source_feedback_to(a)	

Figure 5 The “External Review” static policy example

4.2.2 Instantiation Policies

Although the StaticPolicy construct is able to express some resource allocation restrictions that are verified during modeling time, the Instantiation offer a specific language to define ordering criteria for the dynamic selection of required resource instances. In fact, Policy enactment specialize the default resource instantiation algorithm available in the *APSEE* system: from required resource set in the abstract part of an activity, the proposed instantiation mechanism can automatically suggest an ordered candidate list based on the required resource types and their current (and expected) availability. Thus, Instantiation Policies constitute user-defined extensions to the default algorithm, with specialized restrictions and order criteria applied to specific process components.

Two examples are shown in figure 6: the leftmost example relies on dynamic (*is_late_to_begin* function call) and static (*get_type*) activity firing properties. Both examples as informally described in the respective Description field. The algebraic semantics for Resource Instantiation Policies is in [LIM01].

ID: "Late activity involving customer and meeting room" Name: "Late activity involving customer" Description: "If activity is late to begin and involves customer meeting, then take rooms that contains a beamer, with year utilization rate less than 10% and size smaller than 10m ² . Order them by highest MediumTimeBetweenFailure first" Interface: a: Activity; ApplyToType: "Meeting Room" Conditions: a.is_late_to_begin() and a.get_type() sub_type_of "Meet Customer" RestrictBy: Contains("beamer") getMetric("year_utilization_rate", <, 10) getMetric("size", <, 10) OrderBy: High_MTBF	ID: "Programming printers" Name: "Programming activities require inexpensive printers" Description: "If an activity belonging to the Programming type produce source code, then select inexpensive printers" Interface: a: Activity ApplyToType: "Printer" Conditions: a.get_output_artifacts().any.get_type() sub_type_of "SourceCode" and a.get_type() sub_type_of "Programming" RestrictBy: Cost(<, 10) NotRequires("Color Paper") OrderBy: Low_Cost
---	--

Figure 6 Resource Instantiation Policy examples

5 FINAL REMARKS

The recent AO Paradigm evolution showed that it can be successfully applied to different domains, including agent-based software engineering [GAR01] and formal methods [BLA98]. This text presented an overview on a novel aspect (policy)-oriented approach for software process modeling. Our experience showed that AO Paradigm seems to be a viable way to formally define critical management and enforcement strategies which are widely disseminated in the process patterns literature.

The proposed Process Policies constructs were successfully applied to a number of real world case studies (described in [REI02a] and [REI02b]). This approach improves the modularity on process model descriptions by providing independent formalisms. For instance, the modification on a policy instance changes the correspondent behavior for all the associated process components.

Few works in process modeling field are related to the description of aspects as first order components for process modeling. Perry proposes in [PER97] user-defined Policy descriptions for the Interact PML. Interact is a rule-based language which divides process description in three main parts: object, policy and activity definitions. Huang and Shan

proposal [HUA99] is similar to Instantiation Policy construct, providing a language to automate fine-grain personnel allocation in Workflow projects. *APSEE* Policies distance from the both cited proposals by encapsulating Policies as independently-defined pluggable elements that can be reused across different process models, as a consequence of adopting AO concepts for process modeling.

Both Static and Instantiation Policies were thoroughly used in the composition of a wide variety of process models, including literature-described and those representing new context-specific processes. This experience is summarized in [REI02a]. At a first sight, the use of the Policies construct would have the potential to increase the chance of conflicts. However, our initial experience showed that the Policies model scales well, since the Static Policies construct is able to detect potential conflicting policies in advance [REI02a], while the instantiation mechanism establishes an ordering algorithm that prevents firing of conflicting policies during instantiation time [LIM01].

It is important to note that the lack of a widely accepted standard notation for both AO software and process models constitutes a critical obstacle to the success of current solutions. In particular, we would like to evaluate the recent evolution of the work by Franch and Ribó [FRA99] (on extending UML to represent enactable process models), as it seems to offer a viable path for PML unification. Similar ongoing work in the context of extending the UML notation and semantics to deal with AO constructs would also be welcome.

Since *APSEE* is an active research project, some important model's components are still under implementation. An immediate extension to this work is to investigate the feasibility of using the proposed constructs on heterogeneous enactment and modeling systems – e.g., different PMLs - in order to take advantage of process designers' previous knowledge on different formalisms. Finally, the use of Policies to restrict the automated adaptation of process models to different contexts is currently under investigation [REI02a].

6 REFERENCES

- [BLA98] Blair, L.; Blair, G.S. The Impact of Aspect-Oriented Programming on Formal Methods. European Conference on Object-Oriented Programming (ECOOP'98) Proceedings..., 1998.
- [DER99] Derniame, J.; Kaba, B.; Wastell, D. (Eds.). Software Process. Lecture Notes in Computer Science 1500. Springer, 1999.
- [EST99] Estublier, J. Is a Process Formalism an ADL? Int'l Process Technology Workshop, 1. (IPTW'99). Proceedings... Sept. 1999.
- [FEI93] Feiler, P.; Humphrey, W. Software Process Development and Enactment. Int'l Conf. on the Software Process, 2. Feb.1993.
- [FRA99] Franch, X.; Ribó, J. Some Reflexions in Modeling of Software Process. Int'l. Process Techonology Workshop, 1. (IPTW'99). Proceedings... Sept.1999.
- [GAR01] Garcia, A. et. al. An Aspect-Based Approach for Developing Multi-Agent Object-Oriented Systems. Braz. Symposium on Software Engineering, 15. Proceedings... Rio, Oct. 2001.
- [HUA99] Huang, Y.; Shan, M. Policies in a Resource Manager of Workflow Systems: Modeling, Enforcement and Management. Int'l Conf. on Data Engineering, 15. Proceedings... Mar. 1999.
- [KEN98] Kenens P., et al. An AOP Case with Static and Dynamic Aspects. European Conference on Object-Oriented Programming (ECOOP'98) Proceedings... 1998.
- [LIM98] Lima Reis, C.A.; Reis, R.Q.; Nunes, D. J. Dynamic Software Process Manager for the PROSOFT Software Engineering Environment. Symposium on Software Technology (SoST'98). Proceedings... Sadio/European Software Institute. Sept.1998.
- [LIM01] Lima Reis, C.A. et al. A abordagem APSEE para Modelagem e Gerência de Recursos em Ambientes de Processos de Software. Brazilian Symposium on Software Engineering, 15. (SBES'2001) Proceedings... Oct.2001 (in Portuguese).
- [LON93] Lonchamp, J. A Structured Conceptual and Terminological Framework for the Software Process Engineering. Int'l Conf. on the Software Process, 2. Proceedings... IEEE CS, Mar.1993.
- [NUN92] Nunes, D. Estratégia Data Driven no Desenvolvimento de Software. Braz. Symp. on Software Engineering, 6. (SBES'92) Proceedings... Oct. 1992 (in Portuguese).
- [OST87] Osterweil, L. Software Processes are Software Too. Int'l. Conf. on Software Engineering, 9. Proceedings... IEEE CS. 1987.
- [PAU94] Paulk, M.; Weber, C.; Curtis, B. The Capability Maturity Model. Addison-Wesley Publishing Co., 1994.
- [PER96] Perry, D.E. Practical Issues in Process Reuse. Int'l. Software Process Workshop, 10. (ISPW'10) Proceedings... Jun.1996.
- [PER97] Perry, D.E. Using Process Modeling for Process Understanding. Software Process Improvement. Proceedings... Dec. 1997.
- [PRE01] Pressman, R.S. Software Engineering: A Practitioner's Approach. 5th European Edition. McGraw-Hill, 2001.
- [REI01] Reis, R. et al. Automated Support for Software Process Reuse. Int'l. Workshop on Groupware, 7. Proceedings... IEEE. 2001.
- [REI02a] Reis, R. et al. Automatic Verification of Static Policies on Software Process Models. Annals of Software Engineering. Special Volume on Process-Based Software Engineering. V.14. Kluwer Academic Publishers, Oct.2002 (to appear).
- [REI02b] Reis, R. et al. Towards a Software Process Model to Support the Design of Mobile Computing Applications. World Conference on Integrated Design & Process Technology, 6. Proceedings... Society for Design and Process Sciences, Grandview, TX, USA (to appear).
- [SCH97] Schlebbe, H.; Schimpf, S. Reengineering of Prosoft in Java. Technical Report. Uni-Stuttgart, Germany. Oct.1997.

QCCS: A methodology for the development of contract-aware components based on Aspect Oriented Design

Anne-Marie Sassen¹, Gabriel Amorós¹, Petr Donth², Kurt Geihs³, Jean-Marc Jézéquel⁴, Karine Odent⁴, Noël Plouzeau⁴, Torben Weis³

1. Introduction

QCCS (Quality Controlled Component-based Software development) [1] is an IST project sponsored by the European Commission that will develop a methodology and supporting tools for the creation of components based on contracts and Aspect Oriented Programming (AOP).

Components that have been designed according to the QCCS methodology will have proven properties, which are formally specified in contracts and can therefore be safely applied to build complex systems and services. Also they may be re-used in other situations. Because each component knows its pre- and post-conditions for usage, it is easy to discover situations in which the component is used erroneously. Hence our work results in a methodology and supporting technology which copes with two key problems:

- Enabling the smooth and sound integration of components from multiple independent sources into complex systems and services.
- Supporting the design and creation of new components.

The QCCS project started in November 2000 and will end in March 2003. In this paper we will describe the results achieved in the first half of the project.

2. Technical approach

The QCCS' methodology will complement other existing methodologies and enhance them. In particular, the methodology will focus on non-functional issues for the specification part; and it will be focused on aspect weaving and transformation for the design side.

QCCS uses software architecture models extensively to support its methodology of model weaving and transformation. These models are based on extensions of the UML metamodel [2]. UML has been chosen as the QCCS standard modeling language because of its widespread use in the industry, its extensibility properties and the strong growth of transformation tools in academic research as well as in industrial tool companies.

While UML is a powerful and widespread modeling language, it is nothing more than a notation and therefore must be used within a software development process. Building such a process for quality controlled component construction is precisely one of the main objectives of the QCCS project and therefore QCCS participants have been carefully examining methodological issues.

¹ SchlumbergerSema, Madrid, Spain

² KD Software, Prostějov, Czech Republic

³ Technical University Berlin, Germany

⁴ IRISA, Rennes, France

Ideally the process must support both the application developer which uses components and the component developer which builds them. The Catalysis method [3] has been chosen as an initial methodological framework for the project. This choice is based on the following factors.

1. Components put emphasis on interactions between them and their environment. These interactions are structured as protocols (which can sometimes be complex). These protocols must be subject to modelling with the same expressiveness as for instance for types. In other words, one should spend much more time specifying interactions sequences (for instance as acceptable patterns of operation calls). The Catalysis method provides the protocol models with the same possibilities found usually for types only: protocol abstraction and refinement.
2. Components need strong means to define their properties from the user point of view. One of the most important conceptual tools is the *contract* notion [4], which enables components to describe their abilities in extremely precise terms. A well-known example of this technique is a pre/post condition definition on object operations. Catalysis supports this and adds important notions such as guarantees; they specify properties that must be held true during some operation execution (this is different from object invariants, which must be true only at entry and exit of operations).

To be able to extend Catalysis, one needs to go to the metalevel and extend the standard models.

Since we address two different activities (programming *with* quality-controlled components and programming these components), we need two different powerful metamodels:

- one which allows application developers to use and extend components in any application; this metamodel will be an extension of the UML metamodel;
- one which allows component developers to create new components.

In the next section we will describe the first metamodel.

3. A metamodel for the use and extension of components

3.1. Input and output contracts

The interface description of components provides information about the component's functional properties. However, non-functional properties need to be described as well, in order to be able to reuse the component in a predictable manner. These can be described by contracts [5], and therefore the metamodel will be extended with contracts. All requirements of a component need to be made explicit, also the requirements which a component demands of other components in order to be able to carry out its job. For instance, an e-shop component could have an interface featuring the following method:

```
setDataBase( in db : IDataBase ) : boolean
```

Some written documentation that comes along with the component may explain that the application has to invoke this method first before doing anything else. That means one of the component's interfaces expressed the requirement implicitly. To make it explicit, these requirements may also be modelled by contracts, with the little difference that the component does not offer this kind of contracts: it demands them. Hence, we distinguish between output contracts, which are offered by the component to the outer world, and input contracts on which the component relies.

3.2. Contract Relationships

Components and objects bear different kinds of flexibility. It is generally not possible to derive from a component and to overload some of its methods. However, a component offers means for interface discovery and configuration management through introspection mechanisms. Therefore, a tool can find out at runtime which properties are available and which type they expect; this facility helps the user in configuring the component. Most of these properties can be

altered at runtime, but some are static and have to be set during or before deployment. However, this mechanism does not offer more than a tool supported parameterisation of a component which does neither affect the *offered* nor the *required* interfaces or non-functional properties. To illustrate the inherent problem of this approach we come back to our e-shop component example and we assume that it offers an e-payment interface that is valid only in some situations, for instance only if data exchanges can be safely encrypted over the communication link. Such requirements on a component's environment can be modelled by an input contract. If this required contract cannot be fulfilled then the component should not offer the e-payment interface. This mechanism prevents the developer from making a fundamental design error. Therefore we need an appropriate UML notation to assist designers and design tools in manipulating contracts and contract relationships.

When dealing with quality of service contracts [5] we will discover that a set of contracts may have exclusive-or semantics. That means only one contract can be active at a certain time. These contracts often share common subsets. To ease the modelling we introduced the concept of compound contracts. A compound contract is a composition of other contracts. A composed contract can play two different roles. It is either a required or an offered contract. Another case where compound contracts are useful is the combination of functional and non-functional contracts. For example, some component demands a certain throughput for an SQL interface it is using. By grouping the interface and the non-functional contract in a compound contract, we can express this relationship.

3.3. Contract Selection and Negotiation

Sometimes the contracts can be selected at design time. However, in other cases (especially in QoS-enabled applications) contracts depend on dynamic factors and have to be chosen at runtime. For example, a webcast application that transmits a rock concert over the internet may decide to switch to a contract that does not offer any video but acceptable sound when the bandwidth is no longer sufficient. On the other hand, the e-payment contract may be selected at design time because some communication component ensures that there will be an encrypted link between client and server. Since this document does not aim at presenting mechanisms to select and parameterise the contracts at runtime we refer the reader to our *Management Architecture for Quality of Service* (MAQS) in [6] and [7] as an example for dynamic contract negotiation.

3.3 UML Metamodel Extensions

We propose an extension to the metamodel of the UML 1.4 beta1 specification that allows to model components and contracts as discussed above [8]. Figure 1 shows our new metaclasses and some excerpts from the UML 1.4 beta1 metamodel.

The biggest change is the addition of *Contract* and its subclasses. A *Contract* has the attribute *isOptional*, which is set to *true* if the developer can decide not to accept the contract. Some contracts however are mandatory for the use of the component, so they are not optional. The *isStatic* attribute determines whether the contract has to be selected before deployment or whether selection is possible at runtime. The *Contract* is a superclass of *Interface* and has a *NonFunctionalContract* subclass. The standard *Interface* element is thus redefined in our model.

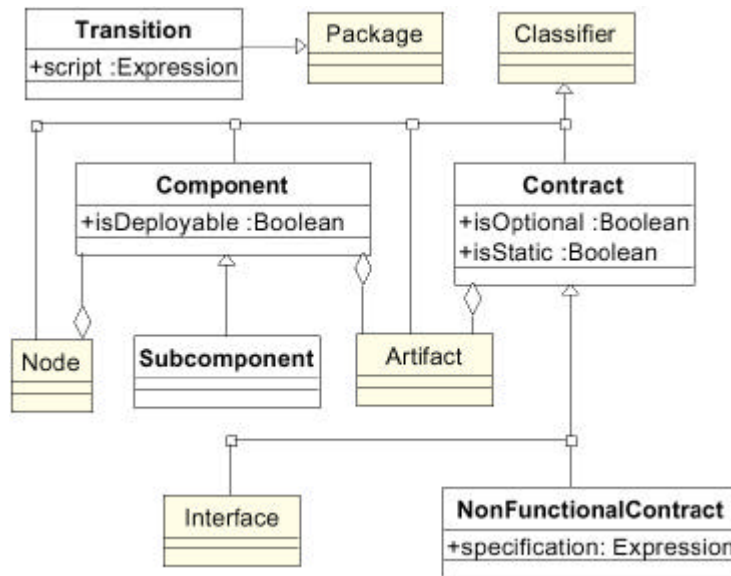


Figure 1. The contract metamodel

- *NonFunctionalContract* instances can be associated with *Interfaces*, which represent functional contracts by grouping them in compound contract as shown in Figure 1. For example, imagine a *NonFunctionalContract* that will deny access to a certain API when the process is running in super-user mode. To model this, one would just draw a *Contract* that has a *Dependency* relationship with the *Interface* and the *NonFunctionalContract*. Connecting the non-functional contract directly with the interface does not work, because that would not allow to model two components requiring the same interface but with different non-functional contracts attached to it.
- *Artifact* is a metaclass that was introduced by UML 1.4 beta1; it is linkable to a contract. In this case, the artifact that represents some physical data will only be installed if its contract has been selected. If the developer decides not to use an optional contract then there is no need to install the data that is used to implement the functionality offered by this contract.
- *Subcomponent* is derived from *Component*, because a subcomponent behaves like a component, but has further restrictions: it cannot be deployed independently. The word *independently* means here independent of its sibling subcomponents. A subcomponent may be connected to its parent component with a *Dependency* relation that has the *trace* stereotype. We now need a way to describe which contract is offered and which contract is required by a component. If a component offers a contract then they are connected by a *Realization* relationship (dashed line with closed arrow). A normal *Dependency* relationship (dashed line with open arrow) between contract and component indicates that the contract is required by this component. The same applies to compound contracts. They can offer and require subcontracts.

3.4. Notation

Our metamodel extension introduces some new metaclasses. Consequently, we have to specify their graphical notation. The UML 1.4 already defines the notation for a component but we have altered it slightly. We added the *isDeployable* attribute to components. If this attribute's value is false then the components' names should appear in italic font. This is still compatible with the current notation since a UML 1.4 component is by definition always deployable. *Subcomponents* are displayed like normal components. The only difference is that the top right corner of its rectangle is cut off. The notation for *Contract* is structurally the same as for classes. That means contracts may have compartments, including but not limited to compartments for attributes and operations. The shape of a contract is not a rectangle. Instead, it resembles the

shape of a convoluted sheet of paper like shown in Figure 2. *Interfaces*, which are in our metamodel a specialization of the contract metaclass, are an exception to this rule. For the sake of compatibility, their notation does not change.

3.5. Example

Let us look at a complete example using the metamodel. We model a simple *Database* component. Figure 2 shows the specification of the *Database* component as given by the components vendor. The database component called “*Component*” realizes two compound contracts (realization is indicated by the closed arrow). One contract combines a *SQL* interface with the non-functional contracts *Availability* and *Thruput*. The other compound contract combines the *Admin* interface with the *Availability* contract. This small example illustrates that different interfaces can be combined with different sets of non-functional contracts.

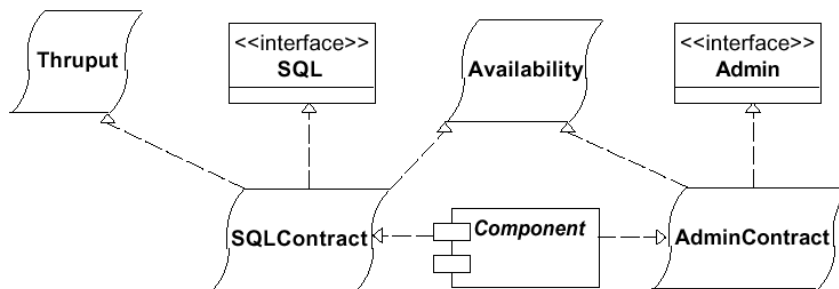


Figure 2. Example of a *database component*

3.6 QML

Our model stores quantitative properties of contracts as QML expressions in the specification attribute of the *NonFunctionalContract* metaclass. The full QML definition is given in the QML reference document [9].

3.7. Using Aspect Orientation for the realisation of contracts

In Figure 2, we do not see how contracts are realised. We just see which contracts are required by a component, and which are offered, and their dependencies. Contracts are modelling non-functional properties of the components. Normally the implementation of these properties are scattered all over the code of the component. Therefore, a useful way to describe the design of the contracts is by using Aspect Orientation. When the designer wants to link a contract (which is an aspect) to a certain component, it means that a new UML model should be generated in which the aspect is woven into the original design. In order to do this automatically with a transformer, the transformer needs to:

1. know where the aspect-specification touches the design (the so-called join points have to be specified)
2. have a weaving algorithm.

At this moment, we believe that an aspect and its join-points may be specified in almost all cases by standard UML, and only in some cases a small extension to UML is necessary. We are also experimenting with a weaving algorithm based on an algorithm of Ullman [10], which is efficient in most cases. More details about this may be found in [11].

The output of this model transformation is another UML model in which the aspect is woven into the design. The designer can continue with the design in the normal way with his standard UML tools, but, he has been able to benefit from reusing an existing design solution. All he needed to do was select a certain standard contract, adapt it to the specific situation, and specify the join points of the existing design with the specific contract.

Of course, since contracts depend on other contracts, several aspects will be woven into the design, resulting in a complex model. However, this doesn't matter, because if for some reason the design of the system needs to be changed, other contracts can be linked to the components, join points may be specified, and a new (woven) design will be generated.

4. Conclusions and further work

QCCS is aiming at a development methodology for contract-aware components. In order to define such a methodology we have defined a meta-model for the contract notion in UML, and we are currently defining how to model contracts in UML using aspect orientation. A tool will be able to interpret contracts and the join points to an existing UML design, and to generate a result design which possesses the required non-functional property. During the project we want to test our new methodology on a workflow system. We have identified mobility to be an important cross cutting concern in the design of our workflow system. The workflow system is a client-server architecture, along the three tier architectural form. A client application is interacting with a business server that manages all workflow related data. Persistency is managed by a relational database that interacts with the business domain server. Mobility has many possible meanings. In the ideal case, a user can work completely disconnected from the network with full read/write access to his/her data. In a more modest and more realistic case, the user can read all his/her data and write part of it. If there is a mobile network involved, the application may provide more of a "normal" service, perhaps with a degraded throughput and/or response time.

The methodology will also be validated on an eCommerce application. There, the crosscutting concern identified was persistency. For both mobility in the workflow system and persistency in the eCommerce system we have specified a standard contract in QML and the metamodel described above. The next step is to model this contract in UML, and to add join points to the current design of the workflow and e-commerce application.

References:

1. QCCS website: www.qccs.org
2. UML Revision Task Force, *UML 14. beta*. 2000.
3. The Catalysis Method. www.trireme.co.uk
4. Meyer, B., *Applying Design by Contract*. IEEE Computer Special Issue on Inheritance and Classification, 1992. **25**(10): p. 40-52.
5. Beugnard, A., et al., *Making Components Contract Aware*. IEEE Computer Special Issue on Components, 1999. **13**(7).
6. Becker, C. and K. Geihs. *Generic QoS Support for Corba*. in *ISCC'00*. 2000. Antibes, France
7. Lorcy, S., N. Plouzeau, and J.-M. Jézéquel. *Reifying Quality of Service Contracts for Distributed Software*. in *TOOLS USA*. 1998.
8. Weis, T., C. Becker, K. Geihs and N. Plouzeau. *A UML Meta-model for Contract Aware Components*, Springer LNCS 2185, 2001
9. Frolund, S. and J. Koistinen, *Quality of service Specification in Distributed Object Systems*. Distributed Systems Engineering Journal, 1998. **5**(4).
10. Ullman, J.R. An algorithm for subgraph isomorphism. Journal of the ACM, Vol 23, No.1, Jan 1976, pp 31-42.
11. Jézéquel, J.-M., N. Plouzeau, T. Weis and K. Geihs, *From Contracts to Aspects in UML Designs*. In *Aspect-Oriented Modelling with UML Workshop*, AOSD 2002.

Early-Stage Concern Modeling

Stanley M. Sutton Jr.

NFA

5 Maple Avenue

Chappaqua, New York 10514 USA

suttonsm@verizonsg.net

ABSTRACT

Concerns are a prominent focus of attention in the early stages of the software life cycle. Particular approaches to requirements engineering and architectural design address particular concerns and relationships, often from a multidimensional perspective. Cosmos is a general-purpose, multidimensional, concern-space modeling schema that can also be used to model early-stage concerns and relationships. Cosmos is not intended to replace other modeling approaches but to supplement and extend them. Cosmos can also be used to bridge the concern gap from early- to later stages of the development life cycle.

1. INTRODUCTION

Aspect-oriented software development (AOSD) attempts to bring the benefits of advanced separation of concerns (ASOC) to the software life cycle. Most work on ASOC to date has focused on the middle, implementation-oriented phases of the life cycle, emphasizing implementation architecture, design, and coding [1,5,4,10,19]. Of course, concerns arise and must be addressed throughout the life cycle, beginning with the early stages of development. Although work in requirements engineering and architectural design has not generally been conducted under the banner of aspect orientation, work in these areas has nevertheless focused quite explicitly on the capture, interrelation, and separation of concerns [20,3,2.12], often with a multidimensional perspective that is consistent with some views of AOSD [19,18].

Given the emphasis on concern modeling and management that can be found in current approaches to requirements engineering and architectural design, it seems that connections to AOSD approaches in later development stages should follow naturally. However, despite the awareness of concerns reflected in these approaches to both early and later stages of development, none of them provides a means for general-purpose concern modeling. Although concerns are heavily modeled, they are modeled mainly in the context of particular tools, methods, or development stages. To capture concerns across the life cycle it is generally necessary to use multiple methods, representations, and models, and the connections between these may be tenuous. Recent work in requirements engineering and architectural design has attempted to span more of the life cycle and provide better connectivity between stages, but the problem is still not addressed in general.

Cosmos has been proposed as a general-purpose concern-space modeling schema [18] that is intended to overcome the limitations of more specialized concern-modeling approaches.

Cosmos provides a framework in which concerns and their interrelationships can be described independently of life cycle stage, development method, and implementation technology.

There are several potential benefits to general-purpose concern-space modeling:

- It can be used to capture concerns in the very initial stages of development, or at other times when they are preliminary or tentative, before they are modeled in requirements, design, or other formalisms
- It can be used to capture concerns that fall outside the scope of other modeling formalisms
- It can be used to relate concerns between different modeling formalisms and development stages
- It can support the modeling and analysis of concerns in the abstract
- It can supplement development tools that may lack a substantial concern-modeling capability

We have used Cosmos independently and with Hyper/J [7] to model the concerns in software components and to support concern-driven component composition [16,17]. Cosmos has also been used for project planning, impact analysis, and development of reusable components [14].

This paper discusses the applicability of Cosmos to the modeling of concerns early in the software life cycle. Section 2 gives an overview of relevant parts of the Cosmos schema, including a running example. Section 3 considers several early-stage modeling approaches, arguing that these reflect considerations appropriate to AOSD and showing with brief examples how they may be addressed in Cosmos. Section 4 provides discussion; Section 5 presents conclusions.

2. COSMOS: A CONCERN-SAPCE MODELING SCHEMA

Cosmos is a general-purpose, multidimensional, concern-space modeling schema. As discussed separately [18], for the sake of generality we consider a *concern* to be any matter of interest in a software system. (This subsumes *aspect* as it has been traditionally defined [9].) We further define a *concern space* as an organized representation of concerns and their relationships.

The Cosmos schema includes three types of elements: concerns, relationships, and predicates. We distinguish between the core schema and extensions to the core schema that may apply to particular installations or applications of the schema.

In this paper we focus mainly on a subset of concerns and relationships, primarily *logical* concerns and *interpretive* relationships (explained below). In describing the schema we give brief examples which are taken from an analysis [16] of a general-purpose software cache, the GPS cache [8]. A goal of this cache is to provide a reusable cache that can be configured

for a variety of applications. The GPS cache has several "top-level" concerns. Some, like core functionality and performance, are typical for caches, whereas others, like generality and management of inter-object dependencies, are more unusual. The Cosmos schema is summarized in Table 1. (Some of the tables and description in this section are abstracted from [18]).

Table 1. Outline of the Cosmos concern-space modeling schema

Core			Extensions
Concerns	Logical	Classifications, Classes, Instances, Properties, Topics	
	Physical	Collections, Instances, Attributes	
Relationships	Categorical	Classification, Generalization, Instantiation, Characterization, Topicality, Membership, Attribution	
	Interpretive	Significance	Admission, Contribution, Logical-Implementation, LogicalComposition, Motivation
	Mapping	Association	Affecting, Description, Modeling, Physical-Implementation, Representation
	Physical	PhysicalRelation	Connection, ConnectionTo, Physically-Affecting, PhysicalComposition
Predicates	<no subtypes defined>		

2.1 Concerns

Cosmos categorizes concerns as *logical* or *physical*. Logical concerns represent the concepts in which we are interested regarding a system or artifact, for example, issues, aspects, features, and properties. Physical concerns represent the system elements or software artifacts to which our logical concerns apply. The two are distinguished because logical concerns can be considered independently of physical concerns and physical concerns can be modeled independently of logical concerns. Both are included because the logical motivate our interests in a system or artifact, whereas the physical allow us to model the system or artifacts in which the logical are realized. Ultimately, it is the relationship between the two on which AOSD depends.

2.1.1 Logical Concerns

Logical concerns represent the conceptual "matters of interest" in a software system. Examples include functionality, behavior, performance, robustness, state, coupling, configurability, usability, size, cost, and so on. Cosmos does not restrict the domain of logical concerns and leaves it to developers (or other stakeholders) to identify concerns of interest. Cosmos distinguishes five types of logical concern: *classifications*, *classes*, *instances*, *properties*, and *topics*.

Classifications represent systems of classes. They are identified with a root class and transitively include the subclasses of that class. From a concern modeling perspective, a classification corresponds to a high-level dimension in a concern space. Because concern-spaces are multidimensional, a particular instance may be concurrently classified according to multiple classifications.

In early work [16?], we categorized concerns in the GPS cache as internal versus external and functional versus non-functional.

External concerns are those seen by users of the system, internal are of concern mainly to developers. Functional concerns include operations and behaviors performed by the system, non-functional include properties and state. All identified concerns were categorized according to both of these dimensions. Subsequently, we identified further classifications, for example, classifications of functionality, of properties, of behaviors, of subsets of behaviors (such as configurability or logging behaviors), and so on. Each of these represents a system of classes of particular types of concern.

Some classifications are independent, such as functional versus non-functional and internal versus external. Others are dependent, such as subclassifications of behavior, which apply to concerns that are already classified as behaviors.

Classifications are important for organizing the organization of concerns. The ability to explicitly discuss classifications makes it possible to explicitly address alternative approaches to the organization and use of concern classes. This is especially important in a multidimensional context, where multiple classifications (and classes) can apply to a given concern and thus where multiple alternative organizations, associations, view points, and access paths for those concerns are possible.

Classes are concerns that are introduced to categorize other concerns. Classes can include other classes (subclasses) and can be used to classify logical instances, properties, and the various kind of physical concern (all discussed below).¹ Classes may be assigned to a classification or may be used apart from any classification.

¹ An implementation of the Cosmos schema in Java is under development that allows classes to apply more generally to all kinds of schema elements.

In the GPS cache, some of the classes of functionality identified include core functionality, functionality related to object invalidation, functionality related to inter-object dependency management, and functionality related to cache configuration. Core functionality (to take one example) can be classified into functions to add objects, delete objects, update objects, and so on. Behaviors can be classified into behaviors to implement functions and other behaviors. Behaviors to implement functions can be classified in parallel with the functions implemented. They can also be classified in other ways, for example, according to performance or concurrency characteristics. Other behaviors include subclasses for logging, buffering, and garbage collection, among others. Alternatively, these can be classified according to whether they may be associated with operations or not and, if so, according to the operations with which they may be associated. (Logging, for example, does not implement any operation but is nevertheless associated with operations, whereas garbage collection neither implements nor is associated with any particular operations.)

Instances are specific concerns that do not classify or characterize other concerns. Instances may be assigned to classes or used independently of any class.

Concern instances in the GPS cache represent specific functions, behaviors, state elements, parameters, and so on. These include, for example, functions to add an object, update an object, and add or update an object; specific behaviors such as those involved in supporting specific functions, in logging operations and statistics, in invalidating out-of-date objects, and in maintaining inter-object dependencies; specific state elements including cache state, object meta-data, and dependency information; and specific parameters such as total cache size and maximum object size.

Properties are concerns that characterize other (logical) concerns. They can be applied to classifications, classes, and instances. Properties applied to classifications apply transitively to classes in the classifications; properties applied to classes apply transitively to members of the class.

Properties are very important in the GPS cache. The primary goal of the cache is to be general. Toward this end, the cache is highly configurable and richly functional. Of course, performance, persistence, concurrency, and recoverability are concerns for many potential applications of the cache, as are correctness of operation and consistency of cached data. The transparency of aspects of the cache implementation is also a concern for its potential impact on users.

Topics are arbitrary collections of concerns. Topics capture theme-related concerns that may belong to several categories. Topics thus provide a way to organize groups of concerns that cut across other Cosmos categorizations. Whereas classifications contain classes of a particular type, and classes contain (sub)classes and members of a particular type, topics may contain elements of any type.

Configurability is a topic we consider important for the GPS cache. This topic includes classes related to configurability (such as configurability functions and configurable behaviors), instances of these classes (specific functions and behaviors, and configurability-related properties). Another topic important for the GPS cache is algorithms, for example, in the areas of efficient dependency management and garbage collection.

2.1.2 Physical Concerns

Physical concerns represent “real world” elements of a system, potentially including software, hardware, systems, and services. As noted above, physical concerns are represented in Cosmos for two main reasons. First, these are the things, such as software artifacts, to which our (logical) concerns apply and through which our logical concerns are realized. Second, these real-world things are of direct concern themselves, not just as derivative or supportive of logical concerns, but as work products, deliverables, and so on.

Cosmos distinguishes three types of physical concern: *instances*, *collections*, and *attributes*. Physical instances represent particular software units, such as .java and .class files. Collections are collections of instances and other (sub)collections, such as packages of code. Attributes are the characteristics of physical instances or collections. (The difference between a physical attribute and a logical property is that the former is a specific characteristic of a specific thing whereas the latter is an abstract concept potentially applicable to many things. For example, performance is a property of interest for the GPS cache, but this general concept is not the same as the specific performance characteristics of specific implementations of the cache.) As physical concerns are not relevant to the examples presented in this paper, they are not discussed further.

2.1.3 Relationships

Cosmos defines four categories of relationship: *categorical*, *interpretive*, *mapping*, and *physical*. The Cosmos core schema defines specific types of categorical relationship, while specific types of the other relationships must be defined according to particular installations or applications.

2.1.4 Categorical Relationships

Categorical relationships relate concerns based on their categories (Table 1). These relationships reflect the semantics of concern types. There are seven types of categorical relationship: *Classification* relates a class to a system of classification. *Generalization* relates classes in an inheritance relationship. *Instantiation* relates instances to classes to which they belong. *Characterization* relates properties to the classes or instances to which they apply. *Membership* relates physical instances to collections to which they belong. *Attribution* relates attributes to physical instances or collections. *Topicality* relates concerns of any type to a topic. These relationships are summarized in Table 2.

Table 2. Cosmos categorical relationships

Element Kind (role)	Categorical Relationship	Element Kind (role)
Classification	Classification	Class
Class (superclass)	Generalization	Class (subclass)
Class	Instantiation	Logical Instance, Property, Collection, Physical Instance, Attribute (member)
Property	Characterization	Classification, Class, Instance
Topic	Topicality	Concern (subject)
Collection	Membership	Physical Instance (member)
Attribute	Attribution	Collection, Physical Instance

2.1.5 Interpretive Relationships

Interpretive relationships reflect interpreted semantic associations among logical concerns. They depend primarily on the context-dependent concern semantics and significance. The Cosmos core schema does not predefine particular interpretive relationship types; these should be added according to concern-modeling needs. In analyzing concerns in the design of the GPS cache, we considered four interpretive relationship types. As indicated in Section 3, relationships such as these are also especially important in early-stage cycle concern modeling. (Indeed, some of these were initially inspired by comparable relationships proposed for requirements engineering.)

Contribution: One logical concern contributes-to another if the way in which, or the extent to which, one concern is addressed affects the way in which, or the extent to which, the other concern is addressed. For example, *optimization* contributes to *performance*. Contribution is not necessarily positive; for example *logging* contributes to *performance* but in a negative way. Contribution is especially important for impact analysis and change propagation. Examples of contribution for concerns in the GPS cache are shown in Table 3.

Motivation: One logical concern motivates another if one concern provides an impetus or justification for the other. For example, *performance* may motivate *configurability* and *recoverability* may motivate *logging*. (Whether these relationships actually hold depends on the particular case.) Motivation supports rationale capture and impact analysis. Some examples of motivation relationships for concerns in the GPS cache are shown in Table 4.

Logical implementation: One logical concern logically implements another if one concern is introduced relative to the implementation of the other. In the GPS cache, a *dependency graph* is introduced to for purposes of implementing *object dependency management*. This is not the same as physical implementation, which would reflect, for example, that a particular code unit implements a particular function.

Table 3. Some examples of contribution relationships for concerns in the GPS cache

Domain	Relationship	Range
Configurability	Contributes-to	Generality
Configurability	Contributes-to (variably!)	Performance
Optimization	Contributes-to	Performance
Logging behavior	Contributes-to (negatively!)	Performance
Logging behavior	Contributes-to	Robustness
Storage optimization	Contributes-to	Optimization
Garbage collection	Contributes-to	Storage optimization
Logging optionality	Contributes-to	Configurability

Table 4. Some examples of motivation relationships for concerns in the GPS cache

Domain	Relationship	Range
Generality	Motivates	Configurability
Performance	Motivates	Configurability
Performance	Motivates (negatively!)	Logging
Recoverability	Motivates	Logging
Performance	Motivates	Optimization
Information hiding	Motivates	Retrieve object-copy function
Performance	Motivates	Retrieve object-original function

Admission: One concern admits another if one concern makes it sensible to consider another. The introduction of *logging* admits *log buffering*. Logging neither requires nor motivates log buffering, but without logging it is meaningless to consider log buffering. Admission reflects the fact that the introduction of some concerns opens the concern space to the consideration of other, semantically dependent concerns.

Some examples of logical implementation and admission relationships for the GPS cache are shown in Table 5.

Table 5. Some examples of admission and logical implementation relationships for concerns in the GPS cache

Domain	Relationship	Range
Logging	Admits	Optionality
Garbage collection AND basic functionality	Admits	Interaction of garbage collection and basic functionality
Cache buffering	Admits	Cache storage allocation
Object dependency graph	Logically-implements	Object dependency management
Object invalidation behavior	Logically-implements	Object dependency management

The interpretive relationship types described above serve particular purposes that were relevant to our objectives in modeling concerns for the GPS cache; others may be defined for other purposes. For example, various kinds of dependencies may be defined for rationale capture among concerns [20].

2.1.6 Physical Relationships

Physical relationships relate physical concerns, for example, reflecting the composition of components into an application. As with interpretive relationships, the core Cosmos schema does not define particular types of physical relationship but allows them to be added according to modeling or analysis needs. The relevance of particular relationships may depend on the kind of physical concerns related, the purpose of the relationship, and on the particular technologies (e.g., development tools) used in working with the physical elements.

2.1.7 Mapping Relationships

Mapping relationships relate logical and physical concerns. *Physically-implements* is an example of a mapping relationship where a physical concern (e.g., a code unit) implements a logical concern (e.g., a function). As with interpretive and physical relationships, the core Cosmos schema defines no particular types of mapping relationships but anticipates that particular types will be defined for particular purposes. Mapping relationships are especially important where logical concerns are used to organize, select, or compose physical concerns (i.e., their corresponding software units) [17].

2.2 Multidimensional Modeling

Cosmos is based on the “multidimensional separation of concerns” premise that concerns in software apply simultaneously in multiple, crosscutting dimensions [19]. Multidimensionality is reflected in several ways in Cosmos. Cosmos defines several kinds of concern, and topics are introduced specifically to capture concerns that cut across the other Cosmos categories. Additionally, Cosmos classifications and classes allow multiple categorizations of concerns. A concern class may be refined according to multiple alternative classifications, a class may be a subclass of multiple super classes, and an instance may belong to multiple classes. Multidimensional concern modeling has proven useful in design and implementation [16,17]; the following examples show that it also applies in early-stage development.

3. EXAMPLES

This section discusses examples modeling approaches from early in the software life cycle. The modeling and interrelation of concerns is already a prominent consideration in these approaches, and many of the notions these approaches incorporate can be readily modeled in Cosmos.

3.1 Preliminary Capture of Concerns

In [16] we describe concerns in the design of the GPS cache. Although this description was formulated after the cache had been implemented, it was based on informal and relatively brief description and discussion of the concerns and their relationships. Thus, in effect this provides an example of the modeling of concerns before they are otherwise formally represented and analyzed, as would occur at the very initial

stages of development. Even in this informal context, we were able to identify a wide variety of concerns and significant semantic relationships among them. As demonstrated by the example in Section 2, all of the kinds of concern recognized by Cosmos came into play, as did several kinds of interpretive relationships (examples in Tables 3-5).

3.2 Requirements Engineering: I* & Tropos

I* is a framework for modeling organizations in terms of actors, goals, and dependencies. Tropos is a methodology that applies this to early requirements and provides a basis for extending early requirements to late requirements, architectural design, and detailed design [20,3].

With respect to concern modeling in requirements, Tropos emphasizes the need to identify organizational concerns, separate these from implementation concerns, and give them first-class treatment. Toward this end, Tropos posits five main classes of concern: actors, resources, (hard) goals, soft goals, and tasks. A particular requirements model will contain multiple instances of each of these classes. Properties are not represented directly in the Tropos schema but may be captured in hard or soft goals (e.g., “increase friendliness of customer service”). Neither are topics (in the Cosmos sense) modeled explicitly, but clearly topics could be useful to modelers, for example, in grouping various instances of the different concern classes according to various application themes (such as all of the goals, agents, and tasks related to customer service).

Tropos (and I*) incorporate several relationships of types that Cosmos would consider interpretive (that is, relating logical concerns according to application-dependent considerations). These include decomposition relationships, means-ends relationships (analogous to Cosmos logical implementation), and dependency relationships (analogous to Cosmos contribution, although Tropos dependencies are ternary whereas Cosmos contribution is binary). Dependencies also relate to the Cosmos notion of motivation.

3.3 Requirements Engineering: KAOS

KAOS [2], like I*, offers a goal-dependency model of requirements and takes a view of concerns that is appropriate to requirements and separated from implementation. Also like Tropos, KAOS provides some downstream continuity, from requirements to architectural refinement.

In some contrast to Tropos, KAOS adopts a more explicitly multidimensional perspective on requirements engineering. This shows up in several ways. Conceptually, requirements in the abstract and elements in a model can be associated with different “aspects” (in their terms) such as “why”, “who”, “when” and “what.” Requirements also have a dual linguistic dimension, including both an “outer” semantic net for general types and semantic relationships and an “inner” assertion language for detailed temporal and logical semantics. Goals can be further classified according to their domain, for example, robustness, safety, efficiency, and privacy. Goals are also subject to disjunctive and conjunctive refinement, with the former providing alternative realizations in an implementation. The KAOS system also supports multiple views of the requirements, including refinement, operationalization, entity-relationship, and agent.

These alternative dimensions of requirements can be represented as classifications in Cosmos. Each such concern classification may have a number of concern classes, such as classes of language (semantic net versus assertions), classes of linguistic constructs (such as agents, goals, constraints, temporal assertions), classes of goals, and so on. As in Tropos, properties per se are not a first class construct in KAOS but are naturally represented by other constructs such as goals and constraints. Neither are topics (in the Cosmos sense) first class constructs, but as with Tropos these could be quite useful with a KAOS model. (To a certain extent, the KAOS views represent the instantiation of particular types of topics.)

As noted, relationships in KAOS include conjunctive and disjunctive goal refinement, which relate to the Cosmos notions of contribution, motivation, and logical implementation. KAOS relationships also include operationalization and responsibility, which reflect further aspects of contribution and motivation.

3.4 Architectural Design: ABAS

ABAS are “attribute-based architectural styles” [11]. These architectural styles are designed to address specific quality attributes and they can be analyzed in terms of these attributes.

ABAS modeling is explicitly multidimensional. It incorporates a number of classifications, including classification of architectural attribute information (in terms of external stimuli, architectural decisions, and responses), classification of architectural elements (in terms of components, connectors, and properties), classification of ABAS specification elements (such as problem description and stimulus/response attribute measures), and classification by property. Additionally, parameters in each of the categories of information are subject to multiple simultaneous classifications. For example, stimuli are classified with respect to mode, source and regularity, and responses are classified with respect to latency, throughput, and precedence. (These are just two of many examples.)

ABAS, in contrast to the requirements methods described, treats properties explicitly and prominently. All architectural styles are classified with respect to the properties of performance, modifiability, and availability. These crosscut the information categories, so that the kinds of information used to describe stimuli, architectural decisions, and responses for performance are different from those used for modifiability. For example, performance responses can be described in terms of latency and throughput, modifiability responses in terms of extent of impact and effort of change.

ABAS does not include the Cosmos topic notion, but (like views in KAOS) certain classifications in ABAS (such as the style specification elements) can be considered specialized instantiations of topics (in that they pull together concerns from various other dimensions).

The sorts of relationships that are emphasized in the requirements modeling approaches (dependency, responsibility) and in Cosmos category of interpretive relationships are not highlighted in ABAS. Of course, ABAS describe kinds of physical relationships (connections) among components that observe various architectural styles. Additionally, ABAS include potentially detailed and quantitative analytical models that describe how specific property measures relate to architectural elements and changes to those elements. Based on

these models it should be possible to derive higher-level associations reflecting dependency, implementation, contribution, and so on.

3.5 Business Architecture

The previous approaches address early concern modeling for application development. Concern modeling is also relevant in other sorts of modeling and development. I have conducted exercises with Cosmos in two other domains, business architecture and development methodology. In both of these cases, Cosmos concepts were generally successful for describing key aspects of the modeled domain.

A business-architecture model was taken from [13]. The described architecture can be modeled in terms of classifications (and classes) including, among others, architecture domains (e.g., business and technology), development paths (e.g., based on existing assets or enterprise knowledge), and business concepts (e.g., business purpose, business situation, and business outcome). The classifications are multidimensional in that certain elements may be classified in multiple ways (e.g., a business role-player may be classified by location, by type, and by capability). Specific instances of these concern classes, of properties, and of topics can be introduced in designing the architecture of specific enterprises. For instance, “customers retained” may be a specific instance of a business outcome, “user-friendliness” may be a property of an automated customer inquiry system, and “customer relations” may be a topic that gathers a number of different properties and concerns from different classes.

The business architecture also includes 19 kinds of relationships that Cosmos would mainly classify as interpretive. Many of these correspond closely to defined Cosmos relationships. Motivation is represented by “motivates”, admission is represented by “defines”, various kinds of contribution are represented by “alters”, “creates”, “governs”, “produces”, and “supports”, and aspects of implementation are represented by “assigned-as”, “enables”, “fulfills”, “performs.” A few of the relationships would be extensions for Cosmos, such as “binds”, “consumes”, and “manipulates.”

The significance of business architecture modeling in Cosmos, as with organizational modeling in I* [20], is that it shows that concerns representing users and the (enterprise) operating environment can be modeled analogously to concerns in the software itself. That in turn suggests that the two may be more effectively integrated both in principle and in practice.

4. DISCUSSION

Concerns are plainly a prominent focus of attention in the early stages of development. Requirements specification and analysis aim precisely to introduce concerns into the life cycle from the user or enterprise perspective; architectural design begins to raise and address concerns related to implementation.

Approaches in both requirements and architecture adopt a multidimensional view of concerns. This is reflected, within individual approaches, in the use of multiple conceptual perspectives, multiple languages, multiple views or specifications, and multiple classifications. Additionally, a multidimensional view of requirements engineering overall has been emphasized by van Lamsweerde [12]. He characterizes the first 25 years of requirements engineering research as

addressing the four dimensions of ontology, structuring, specification, and reasoning, and he argues that the next 25 years will focus on “multi-X” approaches, integrating multiple languages and formats, multiple facets and viewpoints, and multiple modes of reasoning.

Cosmos seems substantially able to represent the kinds of concerns and relationships proposed for early-stage modeling.

- Modern requirements engineering approaches propose specific kinds of concerns (such as goal, agent, etc.). Cosmos does not directly include specific concern kinds like these, but they can be defined using the ontological mechanisms in Cosmos (classifications, classes, and instances). Cosmos is thus on a meta-level with respect to specific concern kinds, analogous to the relationship between UML [15] and specific object kinds (classes).
- Cosmos recognizes properties as an explicit category of concern, which the requirements engineering methods do not. However, these methods do support the modeling of properties in other guises (such as goals), and properties are commonly addressed in specific models. Additionally, properties are a first-class notion in ABAS. Thus, the incorporation of properties into the Cosmos schema seems well justified for early-stage concern modeling.
- The requirements and architectural approaches do not incorporate an explicit notion analogous to Cosmos topics, which support arbitrary grouping and indexing of concerns. These particular approaches may be complete enough in themselves that generalized topics are not needed (although particular kinds of viewpoints and specifications in some of these approaches may be seen as instantiations of specific kinds of topics). Nevertheless, across a broader range of life cycle activities and artifacts, and possibly as a supplement to early-stage approaches, the general notion of topics should still be useful.
- Regarding the interpretive relationships (generally based on interpretation of application or problem semantics), the specific relationships we have previously defined for Cosmos seem to address more or less the same issues as relationships proposed in requirements modeling. However, in particular approaches the particular relationships (e.g., forms of dependency) often vary. Cosmos is intended to be extensible with respect to such relationships so as to accommodate specializations for specific approaches. What is most important for the purposes of Cosmos is that relationships of these types be addressed; the particular varieties of relationship should be tailorable as needed.

Based on the experience with early-stage concern modeling, combined with experience in concern modeling for design and implementation [16,17], it seems that general-purpose concern modeling can address concerns across a significant range of the life cycle, larger than those spanned so far by particular early- (and later-) stage modeling approaches. Additionally, it seems that general-purpose concern modeling may extend and

supplement the kinds of early-stage concern modeling that are now performed, supporting concern modeling in different contexts or for different purposes, and possibly enabling different perspectives on concerns to be captured. However, it is not the intention of Cosmos to replace traditional forms of concern modeling or documentation but to be used with them.

Some of the benefits and applications of Cosmos are due to its relatively general nature compared to specific modeling approaches. Would a scheme that was more general still be even more useful? Approaches such as relational and entity-relational do not embody concepts of concern modeling as first class elements. Cosmos is able to leverage the specialized semantics of that domain. Object-modeling approaches, such as UML [15], have some but not all of the concepts embodied in Cosmos. Their generic mechanisms can be used to represent concern-modeling notions, but not as first class elements. Also, the domain of object-modeling is not the same as that of concern modeling, as concerns are mainly concepts, not objects.

5. CONCLUSIONS

Concerns are a prominent focus of attention in the early stages of the software life cycle, including both requirements engineering and architectural design. Particular approaches in these areas address particular concerns and relationships, often from a multidimensional perspective.

Cosmos is a general-purpose concern-space modeling schema. It represents both logical concerns (“matters of interest”) and physical concerns (systems elements). Logical concerns are categorized in terms of classifications, classes, instances, properties, and topics. Kinds of relationship in Cosmos include categorical, interpretive, mapping, and physical. This general modeling schema is substantially able to represent early stage concerns and to capture or approximate important relationships among these concerns. Cosmos has previously been used to model concerns in the design and implementation stages.

Cosmos is not intended to replace other forms of modeling or documentation but to be used with them. Potential applications include preliminary concern modeling (prior to more formal or specialized modeling), supplemental concern modeling (addressing concerns or perspectives not otherwise covered), basic concern modeling for tools that may lack the capability, the linking of concerns across stages, artifacts, and approaches, and indexing into more comprehensive models and documents. Through such applications Cosmos can enrich the modeling of early-stage concerns and help to strengthen the connections between early- and later-stage concerns and between the activities and artifacts in which these are addressed.

6. ACKNOWLEDGMENTS

For earlier collaboration on Cosmos I thank Isabelle Rouvellou. For discussion on MDSOC and Hyper/J I thank Peri Tarr and Harold Ossher. For sharing his experience with Cosmos I thank Juri Memmert. For help with the GPS cache I thank Arun Iyengar and Lou Degenaro. For additional discussions on MDSOC I thank Stefan Tai and Thomas Mikalsen.

7. REFERENCES

- [1] Aksit, M. Wakita, K. Bosch, J., Bergmans, L., and Yonezawa, A. Abstracting object-interactions using composition-filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, Eds., *Object-based Distributed Processing*, Springer, Verlag, 1993.
- [2] Bertrand, P., Darimont, R., Delor, E., Massonet, P., and van Lamsweerde, A. GRAIL/KAOS: An Environment for Goal Driven Requirements Engineering 20th 22nd Int. Conf. on Software Eng. IEEE-ACM, Kyoto, April 1998.
- [3] Castro, J., Kolp M., and Mylopoulos, J. Towards Requirements-Driven Information Systems Engineering: The Tropos Project, 35 pages. To appear in *Information Systems*, Elsevier, Amsterdam, The Netherlands, 2002.
- [4] Clarke, S., Harrison, W., Ossher, H., and Tarr, P. Towards Improved Alignment of Requirements, Design, and Code. Conf. on Object-Oriented Programming, Systems, Languages, and Applications, Denver, Colorado. ACM SIGPLAN Notices, v. 34, n. 10, pp. 325--339, 1999.
- [5] Harrison, W. and Ossher, H. Subject-oriented Programming (a Critique of Pure Objects). Conf. on Object Oriented Programming: Systems, Languages, and Applications, Sep. 1993.
- [6] Harrison, W., Ossher, H., and Tarr, P. Software Engineering Tools and Environments: A Roadmap. In Finkelstein, A., ed., *The Future of Software Eng.*, 22nd Int. Conf. on Software Eng., Limerick, Ireland; ACM, New York, pp. 263-277. June, 2000.
- [7] IBM. Hyper/J. <http://www.research.ibm.com/hyperspace/HyperJ/>.
- [8] Iyengar, A. Design and Performance of a General Purpose Software Cache. In Proc. of the 18th IEEE Int. Performance, Computing, and Communications Conf. (IPCCC'99), Phoenix/Scottsdale, Arizona, Feb. 1999.
- [9] Kiczales, G., Lamping, J. Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. Aspect-Oriented Programming. European Conf. on Object Oriented Programming, Finland. Springer-Verlag LNCS 1241, June 1997.
- [10] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. Getting Started with AspectJ. Comm. of the ACM, v. 44, n. 10, pp. 59-65. Oct. 2001.
- [11] Klein, M. and Kazman, R. Attribute-Based Architectural Styles. Technical Report CMU/SEI-99-TR-022, Oct. 1999. (<http://www.sei.cmu.edu/publications/documents/-99.reports/99tr022/99tr022abstract.html>)
- [12] van Lamsweerde, A. Requirements Engineering in the Year 00: A Research Perspective. 22nd Int. Conf. on Software Eng., ACM Press, 2000, pp. 5--19.
- [13] McDavid, D. A Standard for Business Architecture Description", IBM Systems Journal, v. 38, n. 1, pp. 12--31, 1999.
- [14] Memmert, J. Employing AOSD Technologies in Large Companies. 1st International Conference on Aspect-Oriented Software Development, Enschede, The Netherlands, April, 2002 (to appear).
- [15] Object Management Group. OMG Unified Modeling Language Specification, version 1.4, Sep. 2001.
- [16] Sutton Jr., S. M. and Rouvellou, I. Concerns in the Design of a Software Cache. Workshop on Advanced Separation of Concerns in Object-Oriented Systems. Conf. on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota, Nov. 2000.
- [17] Sutton Jr., S. M. and Rouvellou, I. Advanced Separation of Concerns for Component Evolution. Workshop on Engineering Complex Object Oriented Systems for Evolution. Conf. on Object-Oriented Programming, Systems, Languages, and Applications, Tampa, Florida, Oct. 2001.
- [18] Sutton Jr., S. M. and Rouvellou, I. Modeling of Software Concerns in Cosmos. 1st International Conference on Aspect-Oriented Software Development, Enschede, The Netherlands, April, 2002 (to appear).
- [19] Tarr, P., Ossher, H., Harrison, W. and Sutton Jr., S. M. N Degrees of Separation: Multidimensional Separation of Concerns. 21st Int. Conf. on Software Eng.. ACM, New York, 1999, pp. 107--119.
- [20] Yu, E. S. and Mylopoulos, J. Understanding "Why" in Software Process Modeling, Analysis, and Design. 16th Int. Conf. on Software Eng. (ICSE 16), Sorrento, Italy; IEEE, 1994, pp. 159—168.