Visualizing Aspect-Oriented Requirements Scenarios with Use Case Maps

Gunter Mussbacher, Daniel Amyot
SITE, University of Ottawa
800 King Edward
Ottawa, ON, K1N 6N5, Canada
email: {damyot | gunterm}@site.uottawa.ca

Michael Weiss
School of Computer Science
Carleton University, 1125 Colonel By Drive
Ottawa, ON, K1S 5B6, Canada
email: weiss@scs.carleton.ca

Abstract

The benefits of aspects and aspect-oriented modelling are beginning to be recognized for requirements engineering activities. However, once aspects have been identified, the behaviour, structure, and pointcut expressions of aspects need to be modeled unobtrusively at the requirements level, allowing the engineer to seamlessly focus either on the behaviour and structure of the system without aspects or on the combined behaviour and structure. Furthermore, the modeling techniques for aspects should be the same as for the base system, ensuring that the engineer continues to work with familiar models. This position paper describes how, with the help of Use Case Maps, scenario-based aspects can be modeled visually and unobtrusively at the requirements level and with the same techniques as for non-aspectual systems. With Use Case Maps, aspects including pointcut expressions are modeled in a visual way which is generally considered the preferred choice for models of a high level of abstraction.

Keywords*: Aspect-oriented Requirements Engineering, Use Case Maps, Scenario Notations, User Requirements Notation*

1. Introduction

Aspects have the potential to significantly influence the future of software development like no other software engineering concept since the emergence of object-oriented techniques. While object-oriented techniques enable complex software systems to be managed from a software architect/designer point of view, aspect-oriented software development presents a view of object-oriented software which is more akin to the requirements engineer's world. Aspects address the issues of scattering and tangling which occurs in object-oriented software because the units of interest to the requirements engineer are fundamentally different from the units of interest in object-oriented software [10]. Note that although this paper focuses on object-oriented techniques, similar arguments in support of aspects can be made for other development paradigms.

Use Case Maps (UCMs) [6][13][15] are part of the *User Requirements Notation* (URN) [2][11], a standardization effort of the International Telecommunication Union (ITU). The Use Case Maps notation is a visual scenario language that focuses on the causal flow of behaviour superimposed on a structure of components. Use Case Maps depict the interaction of architectural entities while abstracting from message and data details. UCMs have been used to drive performance analysis, scenario interaction detection, testing activities, and the evaluation of architectural alternatives at a very early stage in the software development process. Over the last decade, UCMs have successfully been used to model service-oriented, concurrent, distributed, and reactive systems in many application domains. UCMs have also been used for business process modeling. The UCM Virtual Library contains an extensive collection of papers and tutorials where these applications are described [13].

In addition to UCMs, URN contains a second language for goal modeling and the description of non-functional requirements (GRL – the *Goal-oriented Requirement Language* [14]), making it the first standardization effort to address non-functional requirements explicitly. GRL integrates many concepts from the *i** [17] and NFR [8] goal/agent frameworks.

As a first step we endeavour to unify UCMs and aspects by using existing notational elements of UCMs to describe aspect-oriented concepts. Aligning UCM and aspect concepts makes it possible to visually describe aspect-oriented models with UCM models (as long as the aspect models are scenario-based). Fur-

thermore, it allows aspect concepts to be used as first class modeling elements when building UCM models.

Aspects can improve the modularity, compositionality, reusability, and maintainability of URN models. Considering the strong overlap between non-functional requirements and concerns encapsulated by aspects, aspects can help bridge the gap between goals (non-functional requirements described with GRL) and operational scenarios (the UCM scenarios which describe how a goal is achieved). On the other hand, aspects can benefit from a standardized way of modeling non-functional requirements (and therefore concerns) with URN. We are planning to extend this effort to GRL in the near future.

Defining and representing aspects must consider a number of *factors*: a) ideally, there should not be a difference between traditional modeling and aspect-oriented modeling; b) it should be possible to define aspects without influencing the base model; c) employing a single modeling paradigm is preferable over using multiple paradigms (e.g. using graphical and purely textual representations at the same time); and d) for scenario/use-case based aspect techniques to be effective at the requirements phase, the employed technique should be at the right abstraction level where message or data details of interactions are not yet relevant.

Section 2 gives a brief overview of the UCM notation and an overview of modeling techniques for early aspects. Section 3 describes how aspects can be modeled with UCMs at the requirements level. A discussion is found in section 4, together with our conclusions and perspectives in section 5.

2. Background

2.1. Use Case Maps

Use Case Maps (UCMs) are a visual scenario notation for the description of functional requirements and if desired, high-level design. Paths describe the causal flow of behaviour of a system (e.g. one or many use cases). Optionally, paths are superimposed over components which represent the architectural structure of a system (e.g. classes or packages). UCMs abstract from the details of message exchange and communication infrastructures while still showing the interaction between architectural entities. As UCMs integrate many scenarios and use cases into one combined model of a system, it is possible to reason about undesired interactions between scenarios.

The basic elements of the UCM notation are shown in the Fig. 1. A *map* contains any number of paths and structural elements called *components*. *Responsibilities* describe required actions or steps to fulfill a scenario. *Paths* express causal sequences. *OR-forks* (possibly including guarding conditions) and *OR-joins* are used to show alternatives, while *AND-forks* and *AND-joins* depict concurrency. Loops can be modeled implicitly with OR-forks and OR-joins. UCM models can be decomposed using *stubs* which contain sub-maps called *plug-ins*. Plug-in maps are reusable units of behaviour and structure. A stub may be *static* which means that it can have at most one plug-in, whereas a *dynamic stub* may have many plug-ins which may be selected at runtime. A *selection policy* decides which plug-ins of a dynamic stub to choose at runtime. Map elements which reside inside a component are said to be *bound* to the component. A more complete coverage of the notation elements is available in [2][15].

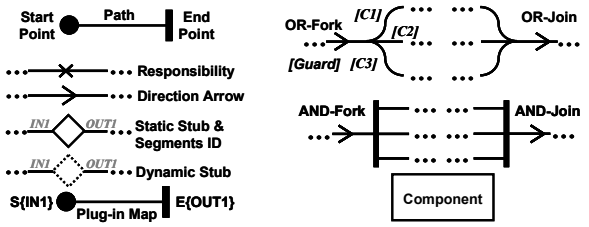

Fig. 1. Basic elements of UCM notation

UCMs share many characteristics with UML activity diagrams but UCMs offer more flexibility in how sub-diagrams can be connected, how sub-components can be represented, and how dynamic responsibilities and dynamic components (not shown here) can be used to capture requirements for agent systems. UCMs also integrate a simple data model, performance annotations, and a simple action language used for analysis. However, activity diagrams have better support for object flows and a better integration with the rest of UML.

jUCMNav is an Eclipse-based tool used to create, maintain, analyze, and transform UCM models [12]. Support for GRL was recently added to enable the editing of complete URN models.

2.2. Aspect-oriented modelling and RE

A decade ago, *aspect-oriented programming* [10] was introduced to address problems with object-oriented software engineering that occur because the units of interest to the requirements engineer cannot readily be encapsulated with object-oriented units. This results in *scattering* (parts of a requirements unit are scattered over many classes) and *tangling* (one class contains parts of many different requirements

units). This problem has also been referred to as the tyranny of the dominant decomposition, as a chosen modularization technique (e.g. objects) inevitably will cause unwanted side-effects in the software design (e.g. scattering and tangling). Examples for requirements units (or *concerns*) for which aspects provide a better encapsulation than objects are authorization and authentication, caching, concurrency management, debugging, distribution, logging, testing, transaction management, or even a feature or use case [9].

Initially, aspect-orientation focused on the implementation level leading to an ever-growing number of tools that provide extensions for aspects to major programming languages [3]. Essentially, aspects identify locations in a program (called *joinpoints*) through parameterized expressions (called *pointcuts*). Aspects also specify behaviour (called *advice*) which will be inserted into the specified locations. As advice may change the behaviour of already existing structural entities, thus violating proper object-oriented modularization, entities external to an aspect are referenced in a structured way (with the help of *intertype declarations*). Note that we are using AspectJ terms [5] but that the concepts also apply to other flavours of aspect-oriented programming.

Aspect-oriented modeling or *early aspects* aims to apply aspect-oriented concepts earlier in the software development life cycle in order to manage more effectively concerns at the requirements and architecture stages. Many approaches for aspect-oriented requirements engineering are described in a recent survey [7], but none of the mentioned scenario/use case-based approaches to aspect-oriented requirements engineering excels in all factors described in section 1. Aspect-oriented UCMs aim to address this shortcoming (see section 4 for a brief discussion).

3. Aspect-oriented Use Case Maps

Unifying aspect concepts with UCM concepts requires the following questions to be answered in the shown order:

- What are good joinpoints in UCMs?
- How is advice specified for an aspect?
- How are intertype declarations specified for an aspect?
- How are pointcuts specified?
- How are pointcuts and advice linked?
- How is the composed system visualized?

We will approach these tasks initially in an informal way. At the end of this section, however, the concepts mentioned above will be defined precisely in the URN metamodel.

3.1. Joinpoint model

In aspect-oriented programming, a joinpoint is a point in the dynamic flow of the program such as a method call, the execution of a method, the initialization of an object, set methods, get methods, or exception handling. For use cases, a joinpoint is defined as a step in a flow of the use case. In UCM terms, this translates directly into responsibilities. Structurally, responsibilities do not differ from any other path elements. Therefore, any path element could be a joinpoint (i.e. a location on the path). Defining any location on the path as a joinpoint gives the most flexibility to the requirements engineer without neither increasing the complexity of aspect-oriented UCMs nor requiring additional modeling constructs. Fig. 2 shows UCM path elements as well as all locations identified by *before* and *after* expressions of joinpoints (small diamonds).

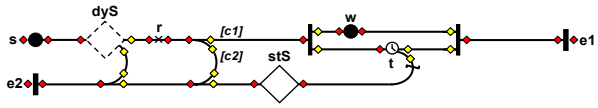

Fig. 2. Joinpoints in UCMs

The small light-shaded diamonds indicate path elements (i.e. joinpoints) with more than one before or after location. The joinpoint model, however, does not need to concern itself with potentially unlimited number of before or after locations. The joinpoint model simply identifies all path elements as joinpoints. We will see in section 3.4 how it is possible to reference each of these before or after locations individually.

3.2. Advice map

An advice describes the behaviour of an aspect triggered in a certain situation. Intertype declarations identify the structural entities that either provide the advice or contribute to the advice. Describing behaviour in UCMs is straightforward as UCMs are meant to do exactly that: describe behaviour with paths on top of a structure of components. The semantic meaning of components in UCMs is cast very wide – anything that can provide a service is a component from classes to actors to roles. Therefore, structural entities identified by intertype declarations can certainly be described with UCM components. The resulting map is called an *advice map*.

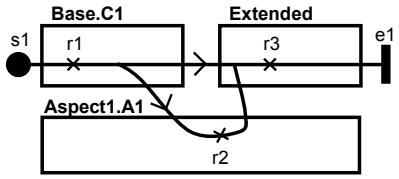

Fig. 3. Advice map with path and components

Fig. 3 shows the description of an advice that requires r1 from an already existing structural entity (Base.C1) in the base, adds component Aspect1.A1 with responsibility r2, and extends an already existing structural entity (Extended) with r3. As the UCM notation in general does not place restrictions on the usage of components, the differentiation between these various types of components is a matter of naming conventions. Note, however, that an advice map does not differ syntactically from a non-aspectual map. Both describe behaviour with a path, and both use components to indicate who is responsible for providing the behaviour. The difference manifests itself in terms of how this map fits into the overall system (explained in section 3.4).

3.3. Pointcut map

A pointcut defines a set of joinpoints either explicitly or in a parameterized way, thereby defining the structural context and behavioural context for the execution of an advice. In aspect-oriented programming, pointcuts can be rather lengthy and complex multi-line expressions. In aspect-oriented use case modeling, pointcuts reference extension points in a different use case. Both use text as the means to describe pointcuts. Considering the visual character of UCMs, a visual representation for pointcuts is a natural choice. Therefore, pointcuts are defined on a UCM called *pointcut map* (see Fig. 4 for four examples).

On first look, there is no difference between a pointcut map and other UCMs. However, the pointcut map represents a partial map which identifies joinpoints when matched against all other maps in the UCM model. The parameterization of pointcuts is achieved by *wildcards* in the names of UCM elements on the pointcut map. Any of the named elements on a pointcut map may contain the wildcard * or logical expressions.

As mentioned previously, pointcut maps are partial maps. This is evident in the usage of *start* and *end* points on the pointcut map. Start or end points without a name denote only the start or end of the partial map and are therefore not matched against start or end points on other maps in the UCM model. Fig. 4 (a) matches all maps with a responsibility starting with Get *and* bound to the component Reservation. Fig. 4 (b) matches against all maps with a responsibility starting with Confirm, followed by an end point confirmed, *and* bound to no component. Finally, the *location* of start and end points is also important for the meaning of the pointcut map. Fig. 4 (c) matches all maps with a responsibility that starts with Set or Get and is bound to any component as the first path element of the component (because the start point is outside the component). Finally, Fig. 4 (d) matches all maps with an OR-fork bound to any component inside component Container as the first path element. The OR-fork is immediately followed on one branch by any static stub and an end point called finished, and immediately exiting the component on the other branch (because the end point is outside the inner component).

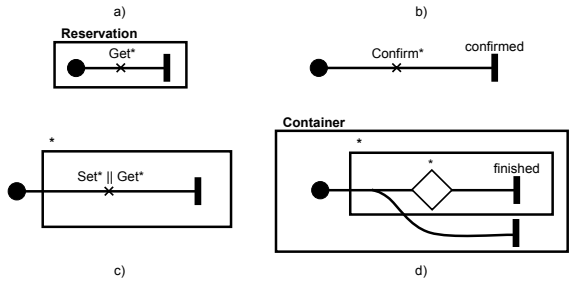

Fig. 4. Pointcut maps

3.4. Advice map revisited

The advice defined on advice maps still needs to be woven into the base system with the help of the pointcuts defined on pointcut maps. Somehow, advice and pointcuts need to be linked. Advice may be executed before, after, or around joinpoints identified by many pointcuts. To this end, the advice map discussed in section 3.2 is slightly changed to include a dynamic stub called the *pointcut stub*. The plug-ins of the pointcut stub are the pointcut maps discussed in section 3.3. For example, the bottom right pointcut map from Fig. 4 could be plugged into the pointcut stubs in Fig. 5 by binding the one in-path to the one start point, one of the out-paths to one of the end points, and the other out-path to the other end point.

The two advice maps in Fig. 5 show how advice and pointcuts are linked to each other. The requirements engineer can understand in one glance the relationship of the advice to the base system due to the visual representation. For example, the top map shows advice being executed before and after the specified pointcuts. It shows that before the specified pointcuts the responsibility Advice.before is executed and after

the specified pointcuts the responsibility Advice.after_returning is executed in the success case and Advice.after_throwing in the fail case. The bottom map shows that the aspect replaces behaviour, a situation which frequently occurs when modeling aspects. The [true] branch is always taken and therefore the responsibility Advice.around is executed instead of the base path elements.

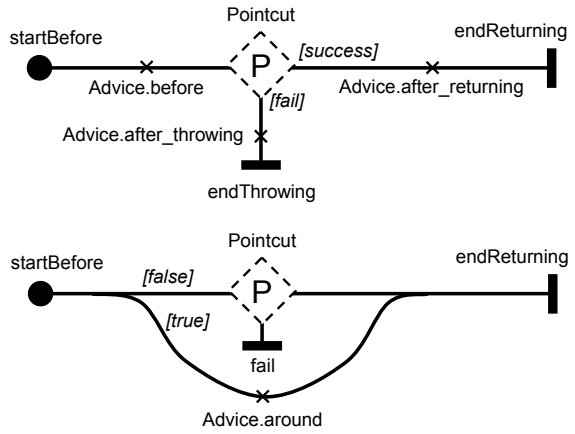

Fig. 5. Pointcut stub

Note that the number of in-paths and out-paths for the pointcut stub is flexible. This makes it possible to take into account path elements with more than one before or after location (e.g., stubs, forks, joins – see section 3.1). Advice can be specified individually for such locations. Also, each advice shown on the maps above is very simple as it consists only of one responsibility. In reality, the responsibility may be replaced by much more complicated advice maps as seen earlier in section 3.2.

3.5. Composed system

Finally, the last remaining question is concerned with how the composed system consisting of the base and aspects is visualized with UCMs. Given the base map, advice map, pointcut map, and plug-in bindings in Fig. 6, the composed system is realized by adding static stubs to the base map which link to the appropriate advice on the advice maps. The locations on the base map where the static stubs are being added are defined by matching the pointcut map against the base map. Then, the plug-in binding information is used to identify which advice needs to be linked to which added stub.

In the composed system, the pointcut stub in the advice map becomes irrelevant and could be either greyed out or even removed if desired. Note that if more than one aspect is adding behaviour at the same joinpoint, the static stub added to the base map becomes a *dynamic* stub, indicating potential conflicts between the aspects.

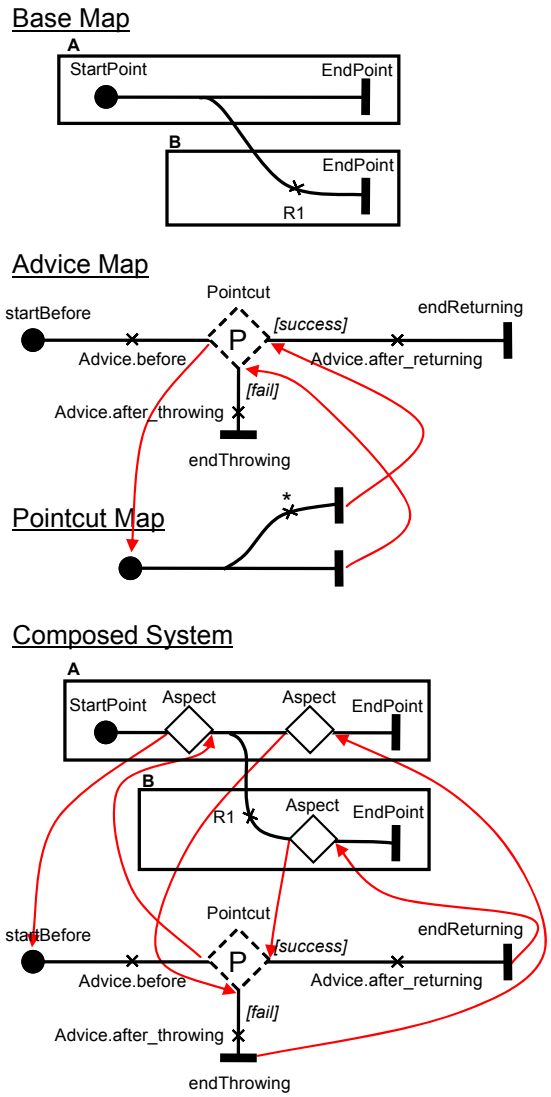

Fig. 6. Composed system

3.6. URN metamodel and aspects

For the purpose of this paper, we are focusing on a subset of the UCM portion of the URN metamodel [12][15], without attributes. A UCMmap consists of component references (ComponentRef) and PathNodes, reflecting structure and behaviour, respectively (see Fig. 7). Components may contain other components as well as path nodes. There are a great number of different kinds of path nodes but only three of them are of interest with regard to aspects: Stub, StartPoint, and EndPoint. Stubs may contain plug-in

maps and PluginBinding specifies how they are connected.

In order to accommodate the new concepts introduced by aspects, the URN metamodel needs to be extended. The new concepts are advice map, pointcut map, pointcut stub, and joinpoint. UCMadviceMap and UCMpointcutMap specialise the UCMmap class whereas PointcutStub specializes the Stub class. Joinpoint is a new class associated with PathNode since each path element can be a joinpoint.

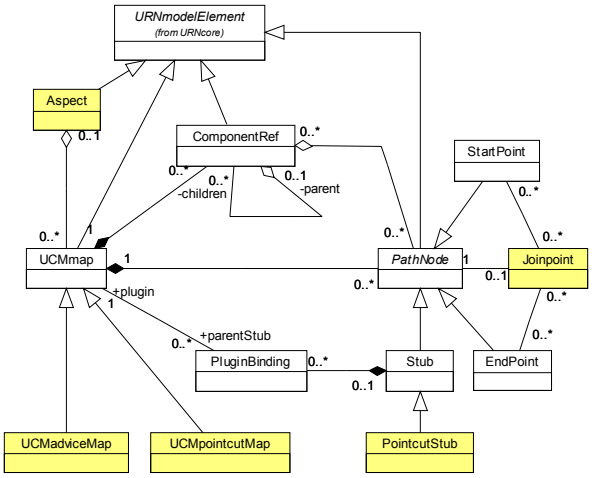

Fig. 7. Extended URN metamodel

With the help of these new classes in the URN metamodel, an aspect can now be defined. An Aspect contains zero or more UCMmaps, some of which are UCMadviceMaps. Only UCMadviceMaps contain zero or more PointcutStubs. A UCMpointcutMap plugs only into a PointcutStub. Finally, only the StartPoints and EndPoints of a UCMpointcutMap match zero or more Joinpoints. These constraints could be expressed with OCL in the aspectual URN metamodel.

4. Discussion

Aspect-oriented UCMs do *not require new notational concepts* but make use of the same set of modeling elements as traditional UCMs, making it easier to switch from traditional modeling to aspect-oriented modeling. Most significantly, stubs are used to link advice and pointcut expression as well as to visualize the composed system. Aspect-oriented UCMs allow aspects to be defined *unobtrusively* without influencing the base model. This is a crucial point of aspect-orientation as the base model must not be polluted by aspect-specific information. Aspect-oriented UCMs model all parts of aspects *visually*, therefore avoiding a modeling paradigm break. Visual models are usually the preferred choice at higher levels of abstractions.

In *Aspect-Oriented Software Development with Use Cases* [9], Jacobson and Ng add the concept of pointcut to use case modeling and change the meaning of extension points (*new concepts*). Furthermore, the base model is *polluted* with aspect information as extension points are defined in the base. Jacobson and Ng use *textual* expression to define parameterized pointcut expressions. Note that the extend relationship for use case diagram is a visual representation of a pointcut but does not contain enough information and therefore has to rely on the textual representation.

Araújo and Moreira [4] require several extensions (*new concepts*) to UML diagrams in order to visualize aspects but do so *unobtrusively*. Their composition rules, however, are represented in a purely *textual* way. Furthermore, these rules explicitly link one element with another and do not allow parameterized expressions.

In *Scenario Modeling with Aspects* [16], Whittle and Araújo do *not require changes* to the used modeling notations (UML sequence diagrams, state machines) and model aspects *unobtrusively*. Their binding rules, however, are represented in a purely *textual* way. Furthermore, these rules explicitly link one element with another and do not allow parameterized expressions.

In terms of abstraction levels, UCMs are at the same level as the use case technique used by Jacobson and Ng. UCMs abstract from message and data details, and can therefore be used earlier than message-based behavioral models but also contain more information than UML use case diagrams. Therefore, UCMs are at a higher level of abstraction than the work by Whittle and Araújo which is at the message/state machine level. Araújo and Moreira make use of some models that are at the same and some models that are at a lower level of abstraction than UCMs.

It is interesting to note that there is also a growing interest in adding aspects concepts to goal models, e.g. the work on aspects and the *i** framework [1]. This makes us believe that aspects can also be added to the GRL part of URN and that this needs to be synchronized with the aspects for UCM scenarios.

Current URN tool support with jUCMNav already allows advice maps, pointcut maps, and pointcut stubs including the binding to pointcut maps to be defined in the UCM model. This does not require any additional features to be implemented as the standard features of jUCMNav are sufficient. Full support for aspect-oriented UCMs, however, requires additional features in order to facilitate the *matching* of pointcut maps to other maps in the UCM model, to indicate on base maps whether a path element is advised, to indicate on advice maps which base maps are being advised by the

aspect, and to switch back and forth between the visualization of the base system and the composed system. A case study of a non-trivial e-commerce application is currently undertaken which we hope will further illustrate the benefits of our approach. As part of the case study, prototype matching and visualization capabilities are being added to jUCMNav.

5. Conclusion

We proposed a scenario-based requirements engineering approach where *one* modeling technique applies to aspects and non-aspectual parts of the system. Base models, advices, and even pointcuts are visually expressed with Use Case Maps, except for the use of wildcards and logical expressions (but this use of text is minimal). UCMs provide a high abstraction level of modelling, above that of many message-based scenario notations, such as UML sequence diagrams, for which aspectual concepts have recently been proposed. The integration of aspects with UCM has a minimal impact on the existing URN metamodel, given that many UCM elements are already amenable to supporting aspect concepts.

This work opens many perspectives. The first one is the integration of aspects concepts for the GRL part of URN and aligning them with those of the UCM part. Combining aspects for goals *and* scenarios at the same time represents a new area of research. Also, it is worthy to note that GRL is mainly about non-functional requirements, just like aspects are to a large extent. Again, we expect some useful synergy to result here.

Regarding tool support, the algorithm that matches pointcut maps with base maps needs to be worked out in detail. We anticipate several issues with scoping when using plug-ins. A composition algorithm is also required, together with visualization features. The current UCM auto-layout mechanism in jUCMNav will be handy, but it will likely require some improvements.

Acknowledgements

This research was supported by the NSERC Discovery program and by the Ontario Research Network on e-Commerce.

References

[1] Alencar, F. *et al.*: Using Aspects to Simplify i* Models. *14th IEEE Int'l Req. Eng. Conf. (RE 06)*, Minneapolis, USA, Sept. 2006.

[2] Amyot, D.: Introduction to the User Requirements Notation: Learning by Example. *Computer Networks*, Vol. 42(3), pp. 285-301, 21 June 2003.

[3] AOSD Community Wiki – *Tools for Developers*: http://aosd.net/wiki/index.php?title=Tools_for_Developers

[4] Araújo, J. and Moreira, A.: An Aspectual Use Case Driven Approach. *VIII Jornadas de Ingeniería de Software y Bases de Datos (JISBD 2003)*, Alicante, Spain, November 2003.

[5] AspectJ web site: http://www.eclipse.org/aspectj/

[6] Buhr, R.J.A. and Casselman, R.S.: *Use Case Maps for Object-Oriented Systems*. Prentice-Hall, 1996.

[7] Chitchyan, R. *et al.*: *Survey of Analysis and Design Approaches*. AOSD-Europe Report ULANC-9, May 2005, http://www.aosd-europe.net/deliverables/d11.pdf

[8] Chung, L., Nixon, B., Yu, E. and Mylopoulos, J.: *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 2000.

[9] Jacobson, I. and Ng, P.-W.: *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley, 2005.

[10] Kiczales, G *et al.*: Aspect-Oriented Programming. *ECOOP'97*, LNCS 1241, pp. 220-242, 1997.

[11] ITU-T: *User Requirements Notation (URN) – Language Requirements and Framework*. ITU-T Recommendation Z.150. Geneva, Switzerland, February 2003.

[12] jUCMNav: http://www.softwareengineering.ca/jucmnav

[13] UCM Virtual Library: http://jucmnav.softwareengineering.ca/twiki/bin/view/UCM/UCMVirtualLibrary

[14] URN Focus Group: *Goal-oriented Requirement Language (GRL)*. ITU-T Draft Rec. Z.151, Geneva, Switzerland, 2003.

[15] URN Focus Group: *Use Case Map Notation (UCM)*. ITU-T Draft Rec. Z.152, Geneva, Switzerland, September 2003. http://www.UseCaseMaps.org/urn

[16] Whittle, J. and Araújo, J.: Scenario Modelling with Aspects. *IEE Proceedings – Software*, Vol. 151(4), pp 157-172, August 2004.

[17] Yu, E.: *Modeling Strategic Relationships for Process Reengineering*. Ph.D. thesis, Department of Computer Science, University of Toronto, Canada, 1995.