Survey of Aspect-Oriented Analysis and Design Approaches

ABSTRACT

A number of Aspect-Oriented (AO) Requirements, Architecture, and Design approaches have emerged recently. In this report we survey the most significant of these approaches, considering their origins, aims, and contributions. Alongside the AO approaches, we also analyse some of the contemporary non-AO work in order to bring out the differences between two sets of techniques, and to understand the potential contributions of aspect-oriented analysis and design.

We also provide some initial insights into processes for AO requirements engineering, analysis and design which may serve as basis for integration of the work of the AOSD-EUROPE project partners. We also outline some issues relevant to such integration.

Document ID:	AOSD-Europe-ULANC-9
Deliverable /Milestone No:	D11
Work-package No:	WP 6
Type:	Survey
Status:	FINAL
Version:	1.0
Date:	18 May 2005
Author(s):	Ruzanna Chitchyan, Awais Rashid, Pete Sawyer, Alessandro Garcia (ULANC) Mónica Pinto Alarcon (UMA) Jethro Bakker, Bedir Tekinerdogan (UT) Siobhán Clarke, Andrew Jackson (TCD)

Table of Authors

Overall Document Editors: Ruzanna Chitchyan, Awais Rashid

Authors: Ruzanna Chitchyan, Awais Rashid, Pete Sawyer

Sections:

1.	Introduction
2.	General comparison criteria
3.	AO Requirements Engineering
3.1	Introduction: Requirements Engineering
3.2	Specific Comparison Criteria
3.3	Non-AO approaches
3.4	AO approaches
3.5	Component-Based AO
3.6	Comparison
6.	Main Contributions of AO Analysis and Design
6.1	Contributions of Aspect-Oriented Requirements Engineering
7.	Emerging AO Analysis & Design Processes
7.1	Emerging Requirements Engineering Process:
8.	Research Agenda to Be Addressed via Integration of Techniques
8.1	Outline of the Road to Integration – Requirements Engineering:
9.	Conclusion

Authors: Jethro Bakker, Mónica Pinto Alarcon, Awais Rashid, Alessandro Garcia, Bedir Tekinerdogan

Sections:

4.	AO Architecture
4.1	Introduction: Architecture
4.2	Non-AO Approaches
4.3	Aspect-Oriented Approaches
4.4	Comparison
6.2	Contributions of Aspect-Oriented Architecture Design
7.2	Emerging Architecture Design Process:
8.2	Outline of the Road to Integration – Architecture Design:

Authors: Siobhán Clarke, Andrew Jackson

Sections:

5.	AO Design
5.1	Introduction
5.2	Specific criteria
5.3	Non-AO Approaches
5.4	AO Approaches
5.5	Comparison
6.3	Contributions of Aspect-Oriented Design
7.3	Emerging Design Process:
8.3	Outline of the Road to Integration – Design

Table of Contents

List of Figures

List of Tables

1. Introduction

With information technology increasingly merging with our everyday environment, the demands on software systems development become more and more challenging. These systems are expected to not only be larger, perform more and more complex functionality, but also to be progressively more reliable, quick and easy to use. Such additional characteristics (often referred to as quality factors) relate to the software systems as a whole hence *crosscutting* their modular structure. Similarly, some functionality (e.g., distribution, synchronisation, etc.) may crosscut several modules in a system. All these make system development ever more complex.

In response to the increasing demands for software development, the Software Engineering discipline has emerged, proposing structured processes and activities to facilitate the development of software. The initial phases of Software Engineering (present in most processes) are Requirements Engineering, Architecture Design, and Design. These phases are the subject of this document. More specifically, we are looking at these phases in the light of a specific Software Engineering methodology: *Aspect-Oriented Software Development* (AOSD). AOSD techniques provide a systematic means for identification, modularisation, representation, and composition of crosscutting concerns [1]. The term *crosscutting concerns* (e.g., reliability, synchronisation) refers to such quality factors or functionalities of software that cannot be effectively modularised using existing software development techniques, e.g., object-oriented (OO) approaches.

Although a significant work has been done in Software Engineering to address complexity of software development, the issue of systematically addressing crosscutting concerns has been overlooked to a large extent. AOSD techniques build on existing work in software development techniques and methodologies in order to tackle such concerns in a systematic fashion. Though most of the initial work in AOSD has focused on developing of aspect-oriented programming (AOP) languages, frameworks and platforms, a number of methods and techniques have also focused on addressing crosscutting concerns at the analysis and design level. Consequently, a significant body of research exists in the area of Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design [2] as well as in modelling and design of systems derived from such aspectual requirements and architectures. The goal of this report is to undertake a survey of the state-of-the-art in handling crosscutting concerns, both in contemporary analysis and design approaches as well as those based on AOSD principles and practice. By this survey we aim to synthesise the relationship between AOSD and relevant non-AOSD techniques at the requirements, architecture and design levels. Furthermore, we aim to elicit initial insights into the roadmap for integrated aspect-oriented requirements engineering, architecture design and detailed design approaches to be developed by the Analysis and Design Lab within AOSD-Europe.

The remainder of this document is structured as follows. In sections 4, 5 and 6 we (respectively) briefly discuss what Requirements Engineering, Architecture, and Design phases are, and what problems they address. We then review a representative set of most prominent contemporary non-AOSD and AOSD approaches within each phase. The discussion on the non-Aspect-Oriented (AO) approaches is not intended as an exhaustive survey: the presented approaches are often selected representatives for groups of approaches relevant to the treatment of crosscutting concerns during analysis and design. Furthermore, in many cases AOSD approaches are extensions of the non-

AO ones. Hence, inclusion of the latter helps to see the contributions of the AOSD methodology more clearly. These contributions are summarised in section 6.

All presented approaches are also discussed with reference to a set of common evaluation criteria: traceability, composability, evolvability, and scalability. The selection criteria are discussed in more detail in section 2 of this document.

In addition to providing the state-of-the-art survey for the AOSD approaches and their contributions, this document is also intended as the basis for development of an integrated Aspect-Oriented Analysis and Design approach, synthesising the work of AOSD-EUROPE project partners. The initial outline of such a candidate process, informed by the earlier discussion of strength, weaknesses, and contributions of non-AO and AO approaches, is presented in section 7. The open issues that need to be addressed in order to realise such an integrated approach are briefly outlined in section 8, pointing to the directions in which further work of the Analysis and Design Lab is likely to progress. Finally, our conclusions for this document are presented in section 9.

2. General comparison criteria

Since AOSD techniques are fairly new, more so at the requirements, architecture and design levels, there are no established metrics or characteristics to compare AOSD approaches and the support they should provide to software engineers. However, a number of such desirable characteristics can be derived from existing best practice in software engineering. Our general comparison criteria is, therefore, based on four qualities that should be facilitated by any analysis and design approach: traceability, composability, evolvability and scalability. These criteria can be further refined for requirements, architecture and design approaches as well as augmented with additional criteria. The refinements and additional criteria, if any, for each level are discussed within sections 4, 5 and 6. Here we provide definitions of our four general comparison criteria:

Traceability through software lifecycle: preservation of traceability between the artefacts of the software lifecycle is one of the crucial qualities required for understandable and maintainable software. Traceability facilitates understanding by relating an artefact to its previous and next representation in an unbroken chain of requirement to code production process, complimented with information for reasons of decisions taken. Thus, once an artefact is conceived (e.g., a requirement is identified), traceability helps to follow it through different representations (e.g., design, coding) and understand the factors/decisions affecting it throughout the development process. This criterion could be broken into two counterparts: (a) ***traceability of artefacts to their source of origin and change*** and (b) ***traceability between lifecycle artefacts***. The first sub-criterion helps to understand where an artefact or a change to it comes from. The second sub-criterion assists with tracking the artefact conversion from one representation and lifecycle stage to another.

Composability: is the ability to compose artefacts and, consequently, to view and understand the complete set of artefacts and their interrelationships, as well as perceive the system as a whole from the integrated artefacts. Composition specifications describe how different analysis and design models are to be composed. In fact, composability is the reverse side of modularity, as only modules that can be composed to work as a united system are of any practical value. While developers might be able to break a system into arbitrary kinds of modules, these will be truly valuable in software development only if composition of such modules can be achieved. Composability criterion in this report implies availability of adequate semantics (e.g., what does it mean to compose manager's and secretary's viewpoints, etc.) as well as syntactic support (e.g., which composition operators to use, how to represent the composed output, etc.). Since AO approaches put forward a new modularisation construct – an aspect – they should also provide new composition mechanisms for it, or show the adequacy of existing mechanisms in dealing with aspectual module.

Evolvability: pertains to the ease of changing the artefacts for an existing requirement/architecture/design, or removal/addition of new ones. An evolvable artefact composition (i.e., the set of system requirement, architecture, or design) will maintain its original conceptual and structural intent in the face of change, without excessive effort on the developers' behalf. This criterion is important because change is an ever present factor in development, and if not supported, will erode the structure of the artefacts, making them difficult to use and maintain, and eventually rendering the software system unusable.

Scalability: is a desirable characteristic because a viable approach should be equally well suited for small and large projects, in particular because projects that start as small ones may grow with time. In some cases the scalability issue can be reduced to tool support. But there are cases where tool support is not sufficient.

3. AO Requirements Engineering

3.1 Introduction: Requirements Engineering

Requirements are always derived from some business problem whether it is, for example, processing passport applications, improving automotive safety systems or adding features to cell phones [3]. Projects may develop new products or they may be concerned with evolving existing or legacy systems. In all cases, the software system will be embedded in an operational context so it will have interfaces to human users, business process elements or other software or hardware systems (Figure 3.1).

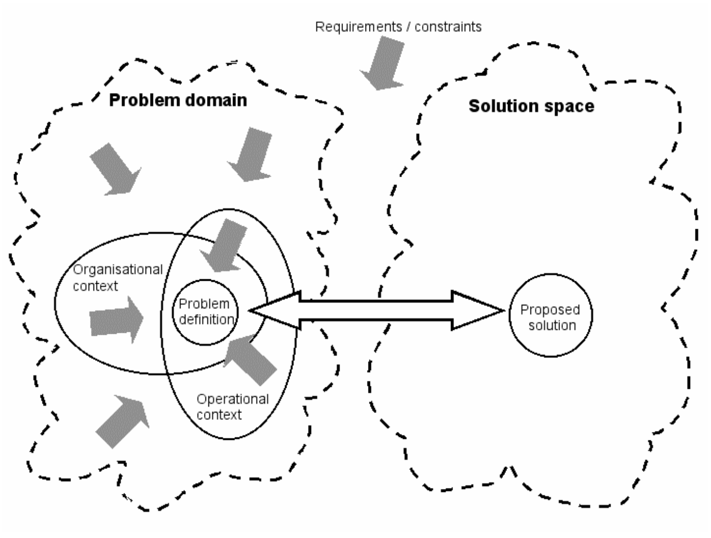


Figure 3-1: Requirements, Constraints, Problems and Solutions in RE

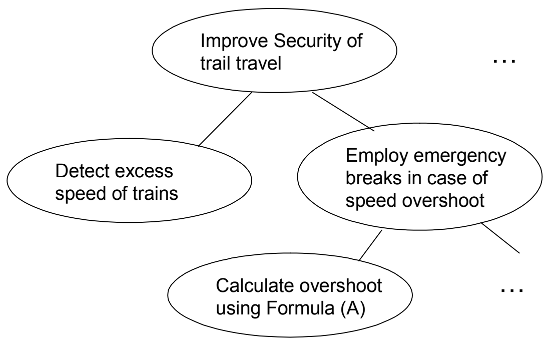
A *requirement* defines a property or capability that must be exhibited by a system in order for it to solve the business problem for which it was conceived.

The classic way to categorise requirements is according to whether they are *functional* or *non-functional. Functional* requirements describe features that the software must provide (for example, "activate the siren when a sensor is tripped"*). Non-functional* requirements, on the other hand, describe qualities of a system. The most important class of non-functional requirements addresses how well the system operates within its environment. Essentially, they specify the quality of delivery of the functional requirements.

Requirements are usually specified at several points on a spectrum that ranges from those with a business focus to those with a technical focus. The technically-focused requirements exist only to make it possible to satisfy the business-focused requirements. For convenience, we will treat these as levels where the highest level requirements are those with a business focus. Such high-level requirements are the goals of the system that set out in very broad strategic terms what is needed to solve some business problem. For instance, in Figure 3-2 the high level goal for some train company is to improve the security of its train travel.

The next level of requirements defines the properties required by the people who will use the system or imposed by other people, organisations or systems within the environment. These are often called the user requirements. In the example in Figure 3-2, the train controller would like to be notified by the system when a train exceeds the set speed for the track section and employ emergency breaks.

One of the main tasks of requirements analysis is to elaborate the user requirements to discover more about the implications of satisfying them. This involves deriving new, lower-level, requirements (*derived requirements*) that focus more on detailed technical issues.



Improve Security of trail travel
…
Detect excess speed of trains
Employ emergency breaks in case of speed overshoot
Calculate overshoot using Formula (A)
…

Figure 3-2 [1]: Levels of System Requirements for a Rail-Travel Security System.

Constraints act to limit the set of possible solutions to the business problem. Some constraints are technical (available bandwidth, for example) while others are related to the problem domain (legislation and standards, for example).

Transforming a requirement into software is a complex process. The deeper into the process, the more design and implementation strategies become committed to satisfying the requirement. Consequently, the costs of rectifying errors in the requirements increase dramatically as development proceeds. An effective RE process which minimises the occurrence of requirements errors and mitigates the impact of requirements change is therefore critical to the success of any development project.

Requirements engineers and developers feel most comfortable when a requirement for a system property can confidently be allocated to a particular software component that assumes responsibility for satisfying the requirement. When such allocation is possible, the resulting software is well modularised, the modules have clear interfaces with each other, and all requirements are cleanly separated.

However, there are many classes of requirements for which clear allocation into modules is not possible using 'traditional' software paradigms. Many non-functional requirements come into this category. For example, performance is a factor of the system architecture and its operational environment; one cannot develop a performance module independent of other parts of a software system. Similarly, it could be hard to allocate responsibility for providing certain kinds of functionality in a cohesive, loosely coupled fashion.

If some requirements are not effectively modularised, it is not possible to reason about their effect on the system or on each other. Furthermore, the lack of modularisation of such properties can result in a large ripple effect on other requirements and their corresponding components upon evolution. The provision of effective means for handling such requirements makes it possible to establish critical trade-offs early on in the software life cycle [5]. Such requirements are termed *crosscutting (or aspectual) requirements*. Examples of such properties include security, mobility, availability and real-time constraints. These properties have a broadly-scoped effect on other requirements or architectural components [5].

[1] Example adapted from [4].

The fact that there is a class of requirements that is hard to isolate within individual modules motivates the work on Aspect-Oriented Requirements Engineering (AORE). The identification and handling of crosscutting requirements is, therefore, the focus of this section of the report. The discussion considers both the body of work on requirements engineering developed prior to emergence of aspect oriented methodologies and that emergent from Aspect Orientation (AO). While contemporary requirements engineering models, e.g., viewpoints [6, 7], use cases [8], and goals [9-11] provide good support for identification and treatment of most kinds of requirements, they do not explicitly focus on crosscutting requirements. The work on aspect-oriented requirements engineering, therefore, complements these approaches by providing systematic means for handling such crosscutting concerns.

Aspect oriented requirements engineering not only aims to provide improved support for separation of crosscutting functional and non-functional properties during requirements engineering, but also to provide a better means to identify and manage conflicts arising due to tangled representations of crosscutting requirements. Such means of early conflict resolution help to establish critical trade-offs even before the architecture is derived [1].

3.2 Specific Comparison Criteria

In order to derive criteria for assessment of Requirements Engineering approaches (in addition to the general comparison criteria for analysis and design approaches we highlighted in Section 2), we turn to the life cycle of requirements and analyse how they arise and progress along the software development lifecycle and the life of the software product. From analysis of the qualities needed for supporting the development and product life cycles, we select the assessment criteria.

3.2.1 Life cycle of Requirements

In section 3.1 we have discussed that during requirements engineering, in broad terms, the properties that the software must exhibit have to be elicited. The analysis of the elicited information and the associated organisational and operational context results in the synthesis of a set of requirements. These requirements need to be, as far as is possible, *correct, complete and feasible*. Achieving these qualities typically requires negotiation and *trade-off* with and between the users and other stakeholders. The set of requirements that emerges from the analysis activity needs to be *recorded* in a specification document that communicates the requirements to the people who will use them to develop the software. The documented requirements need to be *validated* to ensure that the software that they specify will meet the needs of the people from whom the requirements were elicited [3]. As development proceeds, the requirements need to be *managed so that changes are controlled.*

The orthodox view of RE demands that the requirements specification reaches an advanced state of correctness and completeness before subsequent development commences. The rationale for this is to try to anticipate and negate the risk posed by changing, missing or erroneous requirements [3]. In some domains, however, it is impossible to identify all the requirements that will serve to define a product for its expected service life. Even if the initial requirements were complete and correct, it is almost certain that these will change and new requirements will arise during the course of the system use. If the product's environment is volatile, the product's requirements will also be volatile. Thus, *coping with change* is an essential need for a requirements methodology.

Even where the process is optimised to minimise requirements change, some reworking of requirements is inevitable after design and coding has commenced. The RE process normally consumes most effort early in a project, but effort needs to be allocated to *requirements management and coping with change* following sign-off of the requirements specification. Irrespective of the strategy adopted by an RE process the fundamentals remain the same; requirements have to be *discovered, understood, recorded, checked, communicated and managed* [3].

3.2.2 Criteria for Requirements Engineering

The issue of functional requirements discovery has been thoroughly addressed by traditional requirements engineering. As mentioned before, it is the issue of discovery of crosscutting requirements that has not been adequately addressed in the past. However, any new approach should be at least as good as the traditional ones. Thus, we define the first assessment criterion as **identification and handling of both functional and non-functional crosscutting and non-crosscutting requirements.** This criterion addresses three issues:

- It requires that the approaches support non-crosscutting concerns, and in this sense are at least as good as traditional requirements engineering approaches;
- It requires that both functional and non-functional types of crosscutting concerns are supported, as aspects could be of both types;
- It requires that both ***identification*** and ***handling*** are addressed, i.e. it cannot be expected that the requirements will be readily available for us, neither can we expect that the identified concerns can be left without treatment.

The issue of understanding the requirements has several nuances:

- The requirements engineer needs to understand what the system stakeholders want, but this can hardly be assessed in any other way than checking the requirements produced by the engineer with the stakeholders. On the other hand, the architects and designers too need to understand what the requirements engineer has identified. Thus, understanding is closely related to the checking issue, and both of these can be assessed through the **verification and validation** [12] criterion.
- While working to understand the requirements, the engineer will deal with such matters as requirements origin, their mutual influences and conflicting needs, their importance to the stakeholders, and how they contribute to the final system. All this cannot be reflected by one criterion. The matters of requirements origin can be addressed by the previously discussed **traceability** criterion, and the familiar **composability** criterion will help in conflict identification. We additionally select the **Trade-off resolution and Decision support** criterion for assessing the support provided by a requirements engineering approach to the engineer for resolving the identified conflicts.

Hence, the **Verification and Validation** criterion is related to both understanding and checking. It covers two areas: (a) the requirements specification should be verifiable by the system stakeholders (e.g., to ensure absence of side effects due to requirements composition); (b) the architecture and design artefacts and decisions should be capable of being validated against requirements to ensure that the requirements are understood by other system developers.

The **trade-off resolution and decision support** criterion is related to conflict resolution. Conflicts are inevitable between requirements and can arise both from crosscutting and non-crosscutting requirements. This criterion helps to assess how a requirements engineering approach supports conflict resolution. This criterion is also related to the issues of recording and communicating requirements, as the trade-off decisions will need to be recorded and communicated to the other system developers.

The issues of recording and communicating are also partially covered by the **traceability** criterion which records and communicates where the requirements come from and if they can be found again in the architecture and design artefacts. But traceability does not help in communicating the decisions as to how and why a specific requirement must lead to a specific architectural or design decision. This issue is addressed by the **Mapping** criterion.

The support for **mapping RE concerns to following stages** of development is particularly relevant for crosscutting requirements because not all crosscutting requirements identified at the requirements stage will progress into formal design artefacts; some will result in decisions, tradeoffs, or alike [1, 13]. Thus, we need to assess how well the RE approaches support this decision-making. For instance, do they provide clear guidelines for facilitating such mapping?

The issues of understanding, recording, and communicating are also affected by the way that different types of requirements are treated in a requirements engineering approach. If there is a single general treatment process for all types of concerns all three above issues are facilitated, as opposed to having different handling procedures for each kind of requirements (e.g., functional, crosscutting, non-functional, etc.). Thus, the criterion of **homogeneity of requirement treatment** assesses if all types of concerns are treated in a similar way. Of course, this does not imply that all concerns should be treated exactly in the same way; this will not allow for accommodating the differences. What this criteria does suggest, is to have sufficient similarity in the concern treatment to avoid several disjointed sets of representational notations, mapping guidelines, etc.

The issue of managing the requirements and coping with change is similar to that of managing other software artefacts and is addressed by the already discussed criteria of **evolvability**, **composability**, **traceability**, and **scalability**.

Thus, the criteria that will be used for assessment of the requirement engineering approaches (in addition to the general evolvability, composability, traceability, and scalability criteria) are support for:

- *identification and handling of* both functional and non-functional crosscutting and non-crosscutting *requirements*;
- *trade-off resolution and decision support*;
- *mapping requirements to following stages of development*;
- *verification and validation*;
- *homogeneity of requirement treatment*.

3.3 Non-AO approaches

In this section we look at representative work from several classes of Requirements Engineering approaches. These classes have been selected because they are both indicative of the contemporary RE work, and have served as a basis for the emerging AORE techniques (discussed in section 3.4 of this document).

For each selected class of RE work we discuss a few representative approaches, looking at their general method, artefacts, and the process. The discussed classes of approaches are:

- viewpoint-based approaches [4, 14] represented by PREview [4, 7, 15] and Viewpoints and Inconsistency Management [16, 17] work,
- gaol-oriented approaches [9] represented by Non-Functional Requirements Framework [10], I* [18-21] , and KAOS [22],
- problem frames represented by the Problem Frames approach [23-26], and last but not least,
- use case- and scenario-based approaches [27] represented by the Use Cases [8] and the Misuse Cases [28-30] approaches.

3.3.1 Viewpoint-Oriented Approaches

When establishing what is required for solving a given problem, Viewpoint-Oriented Requirement Engineering Approaches consider the problem-related information from different agents (e.g., users of the software system) which can have different, often equally valid, and incomplete perspectives on the problem [4]. These partial perspectives arise due to different responsibilities, roles, goals, or interpretations of the information sources. The combination of the agent and the view that the agent holds is termed a *viewpoint* [14].

When working with large systems that have complex structure and many interlocking constraints on system construction and behaviour, viewpoints assist in understanding and controlling the complexity by separating interests of various actors. Viewpoint-oriented approaches [14] formalise this multi-perspective view into analysis methods. It is because of these multi-perspective views, that viewpoints have been used as a basis for AORE work (presented in section 3.4.1).

3.3.1.1 PREview

PREview [4, 7, 15] is a viewpoint-oriented approach which complements the standard notion of viewpoints with that of *organisational concerns*. A PREview *concern* is a generalisation of the notion of goal; it includes both organisational goals and constraints that restrict the system or process to be analysed. PREview approach was developed to capture such broad requirements as response time, safety, and security.

3.3.1.1.1 PREview Method

A distinctive characteristic of PREview *concerns* is that they “cut across ALL viewpoints and the questions associated with concerns must be linked to all viewpoints and posed to viewpoint sources as part of the analysis process” [4]. Here *functionality* of the system is another concern among these *organisational concerns* and PREview suggests that functionality is often negotiable, as opposed to other concerns (e.g., *safety* in a safety-critical system).

PREview uses *concerns* as drivers in requirement discovery. While using viewpoints for actual requirements discovery, the *concerns* (that are identified at the very start of the RE process and are decomposed into questions, constraints, or requirements) should be addressed by all viewpoints. The decomposed concerns are used to make decisions on concern mapping to functional modules, early architectural or other decisions, etc.

It should be noted that PREview *concerns* are handled differently from *goals* in the goal-oriented approaches: *concerns* are not necessarily refined *through goals* to requirements, but are used as means to identify critical information for requirements elicitation [31] (for instance, by generating questions and constraints that must be addressed by all viewpoints).

Having a firm grounding in 'most crucial concerns' for each project, PREview analysis helps in producing not only a good requirements specification document, but also provides a number of 'side products' that are extremely useful for later stages of software development. For instance, having identified reliability as a concern, it could then be decomposed into a set of functional modules for redundancy (early architectural decisions) and a specification of a verification and validation procedure [4], etc. The process is facilitated by a small set of templates.

3.3.1.1.2 PREview Artefacts

PREview provides templates for concerns and viewpoints.

A concern template is illustrated by means of the *Safety* concern in Figure 3-3, where an on-board train protection and speed control system is used as an example:

Name:	Safety
Description:	Although designated as "critical non-safety software" the on-board train protection system is nevertheless concerned with safety insofar as it must "...contribute to the safety of trains...controlling, in real time, the respect, by the driver, for the operational rules in force on the line."
External Requirements:	ER1, ER2, ER3
Questions:	N/A

Figure 3-3 [2]: Safety Concern in PREview.

Here the name and description sections are self explanatory. The External Requirements section contains references to requirements that represent the concern elaborations that exert a direct influence on the requirements process. In case of conflicts, these constraints from concerns generally override viewpoint requirements [31]. For instance, the safety concern from the above example could be represented as:

> ER1: *The system shall detect the occurrence of excess speed.*
>
> ER2: *The system shall detect the occurrence of overshoot.*
>
> ER3: *The system shall apply emergency braking when either excess speed or overshoot are detected.*

All external requirements above are now straightforward functional requirements. However, in some cases it could be necessary to complement these requirements with questions and/or constraints. For instance, if our train protection system has to link-up with the station safety system we could have needed to consider a question:

> Q1: *Are the requirements compatible with that of the station safety system?*

[2] Source of Figure: [4] section 9.2.

Figure 3-4 provides an example of a PREview viewpoint. Here viewpoint name is complemented by Focus. *Focus* is a statement of perspective adopted by a viewpoint showing how the viewpoint relates to a part or the whole of the system.

The organisational goals and constraints applicable to the system under analysis are listed in the Concerns section of the viewpoint template, while the suppliers of information associated with the viewpoint are provided in the Source section. Records of change history to the viewpoint as well as sources are kept for information traceability. Finally, the requirements section states the requirements of the given stakeholder (i.e. driver in our example) with respect to the train protection system.

Name	Driver
Focus	Usability and ergonomic requirements of driver's interaction with the system
Concerns	Safety
Source	Existing drivers of trains (reference to the drivers names): P. Swift SNCF driver safety regulations (reference to the document): No 123 Driver ergonomics recommendations (reference to the document): No 4
Requirements	• D1 (Driver interaction)
Change history	

Figure 3-4[3] : PREview Viewpoint

Each requirement in turn is represented through a template consisting of a requirement identifier, description, rationale, source, and change history.

Having identified viewpoints and requirements, PREview suggests to use decision tables, based on Quality Function Deployment, to cross-check requirements against overlapping, conflicting, or being independent of external requirements. Such a table is depicted in Figure 3-5. Conflicts between requirements are marked with "-", overlapping among requirements with "+", and independence with "0". Similar tables can be used for detecting dependencies between two viewpoints' requirements.

		External	reqs.	
		er1	er2	er3
VP1	req1a	+	0	0
	req1b	+	-	+
	req1c	0	0	0

Figure 3-5 [4]: Crosschecking Viewpoint 1 (VP1) against External Requirements

3.3.1.1.3 PREview Process

The general outline of the PREview process is presented in Figure 3-6. It starts with establishing high level concerns, for instance, from discussion with senior management. These concerns are then elaborated to a more specific form (e.g., Safety will be decomposed to specific hazards through hazard analysis) and presented as specific requirements (e.g., *external requirements* [7]), constraints or questions. The questions help to collect essential information from each viewpoint's perspective.

[3] Source of Figure: [4] section 9.2.
[4] Source of Figure: [4] section 6.2.3

After concerns have been elaborated, the viewpoints have to be identified from analysis of the applications. PREview provides a standard viewpoint hierarchy of interactors, stakeholders, and domain phenomena which can be used as a starting point for viewpoint identification.

This is followed by requirement discovery for each identified viewpoint. At this stage it is essential to apply the questions and requirements derived from the concerns to each viewpoint. After requirement discovery viewpoint interactions have to be analysed using decision tables. This analysis process is somewhat assisted by viewpoint focus.

Once identified, viewpoint inconsistencies need to be resolved through requirement negotiation and some trade-off mechanism, however this stage is not directly supported by the PREview process.

The process completes by producing the requirements definition document.

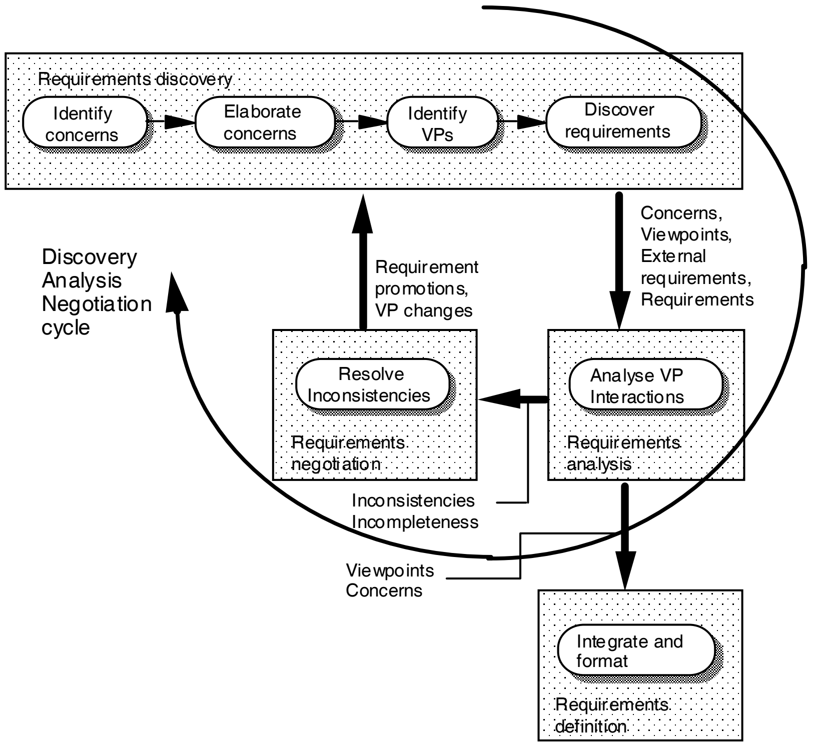


Figure 3-6 [5]: The PREveiw Process Model

Some of the outstanding problems in PREview are:

- Only a small number of concerns (no more then 5) can be effectively addressed in each project. Larger number of concerns makes the amount of generated information unmanageable;
- Similarly only a small number of viewpoints (under 6) should be used;
- Absence of clear guidelines for concern decomposition;
- Absence of a mechanism for inconsistency management, trade-off analysis and decision support.
- Though *Functionality* may be a concern, no examples of functional concerns are provided, raising the questions: can this approach really identify functional crosscutting concerns?

[5] Source of Figure: [4] Figure 6.1.

3.3.1.1.4 Identification and Treatment of Crosscutting Concerns with PREview

The PREview approach is structured around recognition of the importance of the impact of non-functional crosscutting concerns. These concerns are identified using interviews with the stakeholders at the very offset of the development and are modularised in concern templates. The concern impact on other concerns, however, is not modularised: answers to concern-related questions and external requirements are spread across each affected viewpoint.

The PREview concern identification and treatment for non-functional concerns is appropriate only when a small stable set of well defined non-functional requirements are involved. This indeed is the type of development that PREview was intended for. The approach will not be suited to volatile domains.

Also, though PREview suggests that Functionality may be a concern, in the same way as the non-functional concerns, there is no demonstration of how such concerns are identified and treated. Consequently, we conclude that the method does not support identification and treatment of crosscutting characteristics for functional concerns.

3.3.1.2 Viewpoints and Inconsistency Management (VIM)

This approach suggests use of viewpoints as a mechanism to handle the inconsistencies between partial requirement specifications.

3.3.1.2.1 Viewpoints and Inconsistency Management Method

In [16] a framework for expressing relationships between multiple viewpoints in requirement specification is defined. Here a viewpoint is a "loosely coupled, locally managed, distributable object which encapsulates partial knowledge about a system and its domain, specified in a particular, suitable representation scheme, and partial knowledge of the process of development" [16]. Thus, the framework allows specification of a system from multiple viewpoints without having to pre-define a prescribed notation to be used for all specifications. Instead, different (heterogeneous) viewpoints can define their own "templates" for notation and processing method. A template can then be instantiated for many viewpoints.

The viewpoints that describe a system should be integratable to allow for that system's representation to be complete. For this, the framework provides meta-integration support: a set of viewpoint consistency rules for viewpoint templates should be defined.

However, the inconsistencies cannot always be immediately resolved. Thus, the system provides support for inconsistency management [32], i.e. support for dealing with inter-viewpoint rules and situations when the rules between viewpoints do not hold [17]. Moreover, in some cases the inconsistencies do not have to be resolved if the cost of their resolution is greater then the risk of tolerating them [33].

3.3.1.2.2 Viewpoints and Inconsistency Management Artefacts

The artefacts of primary importance in this approach are the viewpoints. They are structured as encapsulations of five information items, as depicted in Figure 3-7.

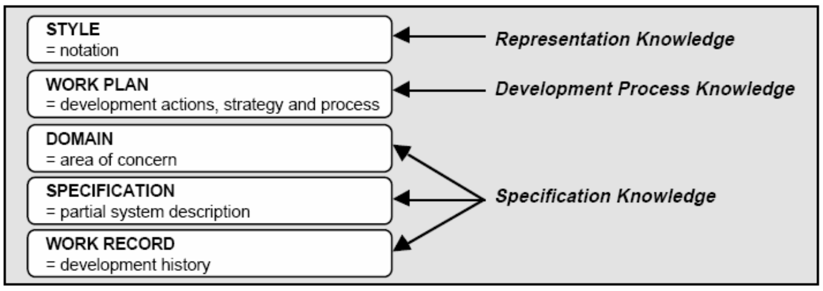


Figure 3-7 [6]: The Five Slots of a Viewpoint.

The top two slots (*style* and *work plan*) of a viewpoint are general for each *viewpoint template*. The *Style* slot describes the notation chosen for viewpoint representation; *work plan* details the development process for the local viewpoint; and a *viewpoint template* is "a reusable description of a development technique (notation and process)" [16]. A template can then be instantiated with a *domain, specification* and *work record*, creating as many instances of a viewpoint template as required. *Domain* identifies the area of the viewpoint focus with respect to the system. *Specification* and *work record* respectively describe the viewpoint domain using the notation and work plan defined in previous sections, and the development history record.

An example of a template and an instance for viewpoints is provided in Figure 3-8 (a) and (b) respectively.

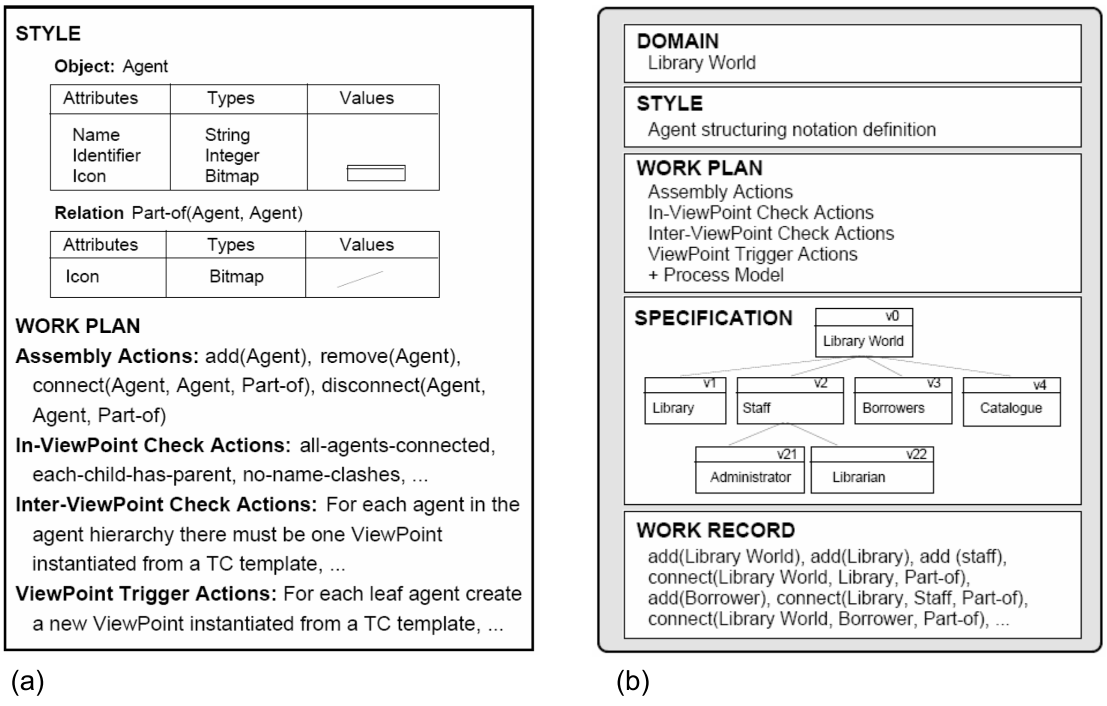

STYLE
Object: Agent
Attributes | Types | Values
Name Identifier Icon | String Integer Bitmap
Relation Part-of(Agent, Agent)
Attributes | Types | Values
Icon | Bitmap
WORK PLAN
Assembly Actions: add(Agent), remove(Agent), connect(Agent, Agent, Part-of), disconnect(Agent, Agent, Part-of)
In-ViewPoint Check Actions: all-agents-connected, each-child-has-parent, no-name-clashes, ...
Inter-ViewPoint Check Actions: For each agent in the agent hierarchy there must be one ViewPoint instantiated from a TC template, ...
ViewPoint Trigger Actions: For each leaf agent create a new ViewPoint instantiated from a TC template, ...
(a)
DOMAIN
Library World
STYLE
Agent structuring notation definition
WORK PLAN
Assembly Actions
In-ViewPoint Check Actions
Inter-ViewPoint Check Actions
ViewPoint Trigger Actions
+ Process Model
SPECIFICATION
v0 Library World
v1 Library
v2 Staff
v3 Borrowers
v4 Catalogue
v21 Administrator
v22 Librarian
WORK RECORD
add(Library World), add(Library), add (staff), connect(Library World, Library, Part-of), add(Borrower), connect(Library, Staff, Part-of), connect(Library World, Borrower, Part-of), ...
(b)

Figure 3-8 [7]: (a) A Viewpoint Template; (b) An Instantiation of a Viewpoint Template for a Library World Domain.

The viewpoint templates can be configured into a structured collection which together create structured steps and notations for system specification, i.e. a configurable (per project) *development method*.

[6] Source of Figure: [16] Figure 1.
[7] Source of Figure: [16], (a) Figure 2; (b) Figure 5.

Another central artefact to this approach is the set of *consistency rules* necessary for viewpoint integration and inconsistency management. Examples of such rules are definition of mapping of notations between viewpoints, or rules defined under the *work plan* section in Figure 3-8 (a), etc.

3.3.1.2.3 Viewpoints and Inconsistency Management Process

The development process in this approach starts with creating the viewpoint templates (i.e. defining the notations and the work plans desired for building a given system). There are no prescriptions as to what notations and work plan should be chosen, the developer is free to choose whatever is best suited to the problem in hand.

Once the templates are developed, they need to be configured into a methodology. In fact, the methodology will emerge from the "local" work plans and notations of defined viewpoints and the consistency rules defined between them while configuring them together.

After this, the produced method can be applied to a specific problem, i.e. the templates can be instantiated using the local notations and work plans, and the consistency rules checked, and inconsistency between viewpoints managed.

The framework for managing inconsistency is presented in Figure 3-9. Its central component is the set of already specified consistency rules. These rules are refined and the set is extended as the development progresses, or new viewpoints are defined.

As the viewpoints evolve, the consistency rules are monitored for each changed/added viewpoint. When a consistency rule is broken, an inconsistency is detected and diagnosed: it is located (where was the rule broken?); identified (reviewing the sequence of actions that led to inconsistency, finding its cause); and classified (in terms of type of broken rule, type of action that caused it, the scale of impact of the inconsistency, etc.).

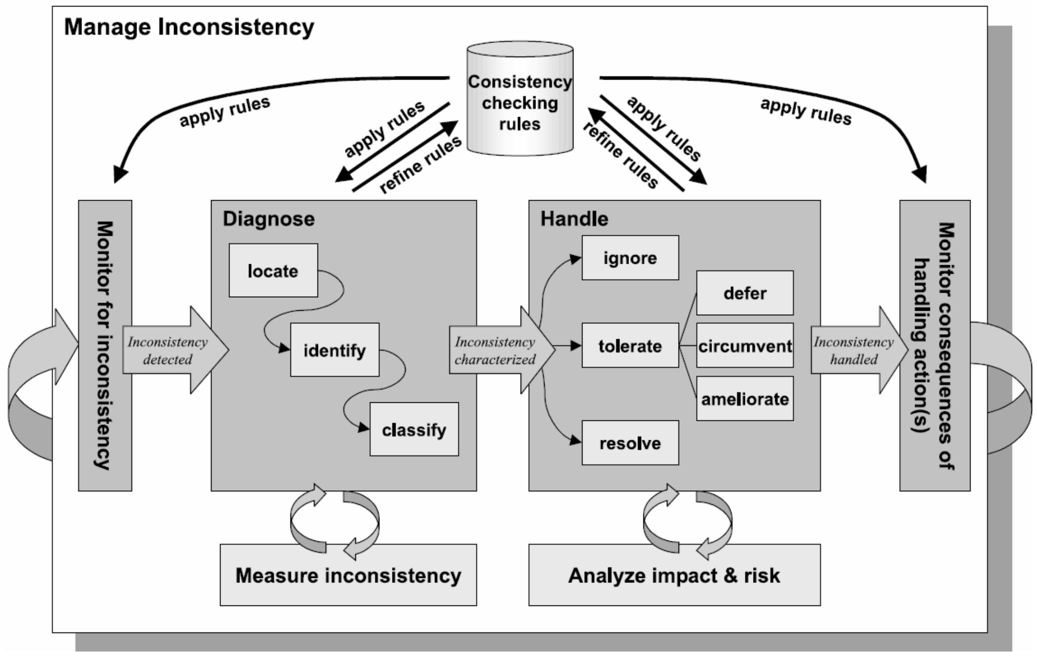


Figure 3-9[8] : A Framework for Managing Inconsistency With Viewpoints.

[8] Source of Figure: [33] Figure 1.

The inconsistency, characterised at diagnostics, is then handled, using one of the available strategies (i.e., ignored, tolerated, or resolved). The chosen handling strategy depends on risk and cost assessment. Whichever cost is lower (e.g., handling vs. ameliorating vs. ignoring), that strategy is recommended.

3.3.1.2.4 Identification and Treatment of Crosscutting Concerns with Viewpoints and Inconsistency Management

This approach does not identify any crosscutting concerns separately: all concerns are treated as part of a viewpoint. Though viewpoints can overlap and crosscut each other, this is treated as an inconsistency resolution issue, rather than concern separation problem.

3.3.1.3 Summary of Viewpoint-Based Approaches

The PREview and VIM approaches discussed above demonstrate how viewpoints are used to focus on the early requirement elicitation stage and later consistency management stages respectively. In fact, these approaches can be used complementarily, with PREview applied to identify the viewpoints and VIM used to manage potential inconsistencies.

Both VIM and PREview fall sort of effectively handling crosscutting concerns, in that PREview limits to a small number of pre-defined non-functional concerns with no support for further concerns, while VIM treats all concerns as part of viewpoints, disregarding their individual modularisation issues. In both cases crosscutting concerns end up scattered and tangled with viewpoints.

3.3.2 Goal-Oriented Approaches

A *goal* is an objective the system under construction should achieve [9]. Goals represent intended properties and can cover both functional concerns that the new system should provide, and non-functional concerns related to its quality of service, such as security, safety, etc. Unlike a requirement, achievement of a goal might require cooperation among multiple agents.

Goals can be used for providing rationale for requirements, assessing requirement completeness and relevance, as well as requirement identification. For instance, a requirement is justifiable and relevant if it leads to satisfaction of a goal, and requirements are complete if all goals are satisfied with the set of defined requirements.

3.3.2.1 Non-Functional Requirements Framework (NFRF)

The NFR Framework [10] is intended for representing and analysing non-functional goals. Central to the framework is the concept of *softgoal* which represents a goal that has neither a clear-cut definition nor precise criteria for determining whether it has been satisfied. Softgoals are used to represent the non-functional requirements. In this respect the definition of softgoal fits quite well, as reliability, for instance, can have different meaning for different people, and even for the same person working on different projects, and so will affect the criteria for defining the level of system's reliability.

The softgoals are interdependent on each other. These dependency relationships are used to see how a softgoal is satisfied, given that some other softgoals are satisfied or denied.

3.3.2.1.1 NFRF Method

The NFR framework consists of 5 major components [10]: softgoals, interdependencies, evaluation procedure, methods, and correlations.

Softgoals are used for representing non-functional requirements. There are 3 types of softgoals: *NFR softgoals*, *operationalising softgoals*, and *claim softgoals*. The *NFR softgoals* act as overall constraints on the system. They are satisfied via *operationalising softgoals* that represent more concrete design or implementation solutions (e.g., operations, processes, data representations, etc.) obtained as a result of decisions made for the NFR softgoal. Operationalisations provide the design alternatives available for the given NFR solution. *Claim softgoals* provide rationale for design and development decisions. Through *claims* domain characteristics can be reflected in the decision making process, particular choices explained, support for prioritising certain softgoals over others provided, etc.

When softgoals are refined *offspring* softgoals are created which relate to their parents through an *IsA relationship*. The offsprings also *contribute* to their parents either positively, or negatively. In the case of positive contribution, satisfying the offspring leads to satisfaction of the parent too, while in the case of negative contribution it leads to dissatisfying the parent. The framework provides a set of refinement methods for different types of softgoals: *decomposition methods* contain the knowledge of how to break softgoals into more detailed ones; *operationalisation methods* assist in operationalising a softgoal, and *augmentation methods* help to represent additional information about a softgoal.

The NFR softgoals, their refinement, and their operationalisations, complemented with claims, together build the Softgoal Interdependency Graph (SIG). The *evaluation procedure* is applied to the SIG to determine the degree to which the initial NFR softgoal is satisfied with the given set of decisions. After NFR requirements have been decomposed, operationalised and evaluated, the NFR SIG is related to the appropriate functional requirement via *design decision* links and the operationalisations are related to the specifications (design decisions) via *operationalisation* links. Thus, when turning to the design stage, the functional requirements have clear links to their related non-functional ones.

The *refinement methods* component of the NFRF provides a set of generic procedures for refining a softgoal or an interdependency into one of more offsprings. These methods are simply patterns (or templates) and guidelines for decomposing softgoals and interdependencies into sub-softgoals and sub-dependencies, based on requirements engineers' past experience and domain knowledge[9].

The non-functional requirements can conflict (e.g., cost and quality) or support (e.g., availability and dependability) each other. These relationships between non-functional requirements (as well as the requirements and operationalisations, and between two operationalisations) are called *correlations*. *Correlations* are used to examine the cross-impact of the softgoals during trade-off analysis.

[9] One can perceive these as loosely similar in role to that of "design patterns" for design derivation.

3.3.2.1.2 NFRF Artefacts

In the NFR framework, the identified *softgoals* need to be catalogued and arranged into types, and hierarchies of *IsA* relationships that refine the initial softgoal. These catalogues are intended for future reuse and to guard against omitting important concerns. Such a catalogue for an information security softgoal [34] is depicted in Figure 3-10. The type catalogue focuses on different type of security (on the right) and their characteristics (on the left). While types are softgoals on their own, characteristics only modify and specialise the meaning of these types.

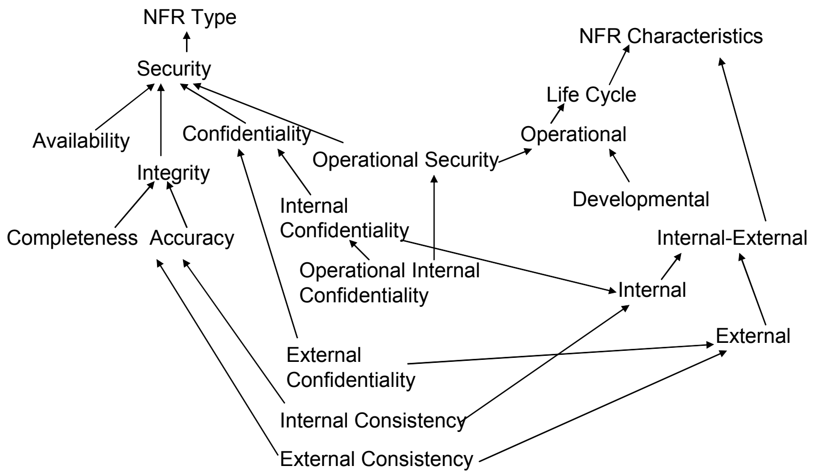


Figure 3-10 [10]: Information Security Type Catalogue in the NFR Framework

The type catalogue and the decomposition methods are used to decompose a specific softgoal. For instance, in Figure 3-11, the account security softgoal is decomposed using the *subtype* method.

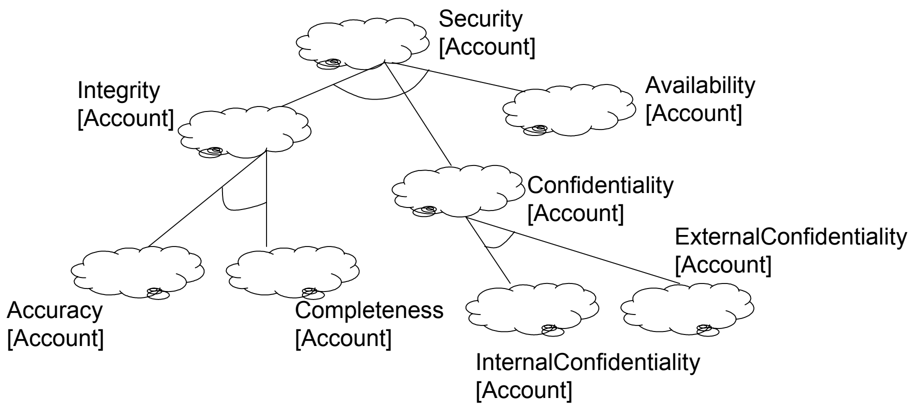


Figure 3-11 [11]: Refinement of Security Softgoal by Subtype

The decomposed softgoals can now be operationalised and augmented to produce an even more elaborate SIG. As illustrated in Figure 3-12, the goal of Internal Consistency is operationalised via access authorisation which is decomposed into authentication, access rule validation and identification sub-goals all of which should be satisfied in order to satisfy access authorisation, as they are defined with an AND operator (the single curve between sub-goal edges). At the lowest level of decomposition the use of single password and card key for authentication has been selected (ticked clouds), leaving out biometrics. We can evaluate that use of single password positively

[10] Source of Figure: [10] Figure 7.1
[11] Source of Figure: [10] Figure 7.3

contributes to password protection of the account. Also password, identification and access rule validation all positively contribute to the access authorisation which in turn positively contributes to internal consistency (the plus signs at the edges).

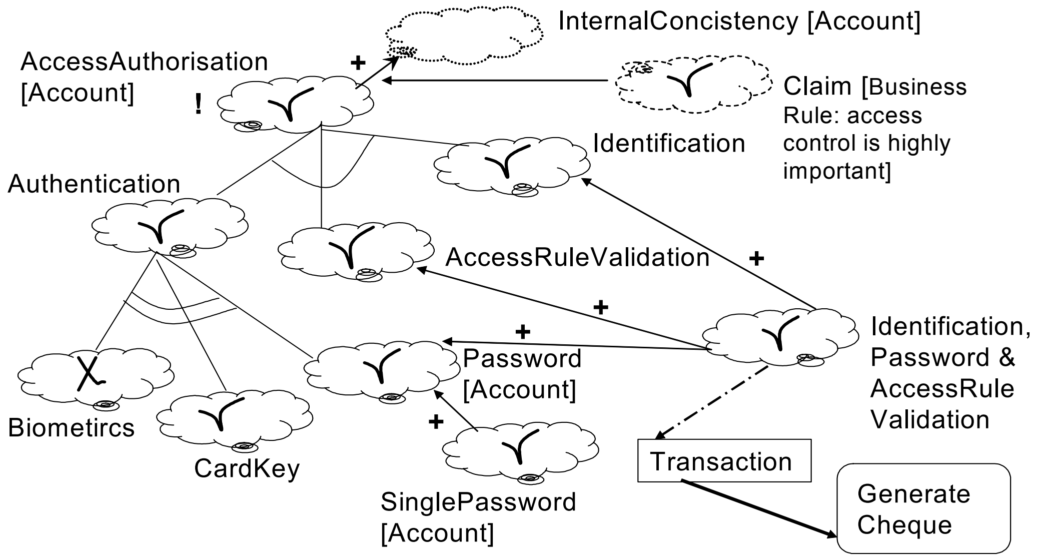

InternalConcistency [Account]
AccessAuthorisation [Account]
+
!
Claim [Business Rule: access control is highly important]
Identification
Authentication
AccessRuleValidation
+
+
+
Identification, Password & AccessRule Validation
Password [Account]
+
Biometircs
CardKey
SinglePassword [Account]
Transaction
Generate Cheque

Figure 3-12 [12]: Partial Software Interdependency Graph for Internal Consistency of an Account

The softgoal satisfaction varies in degree between *satisfied* to *denied*. The degree is determined by evaluating the decisions about offsprings' satisfaction and contributions to the parents. The offspring satisfactions are propagated to parents through their contributions and their interrelationships, but also designer's involvement is required to make judgement in uncertain cases. During decomposition the offsprings can be given different priorities and these *priorities* can be used while making decisions about trade-offs and resolving conflicts between softgoals.

The selected operationalisations are related to the target system which originates from functional requirements (e.g., the Transaction target originates from Generate Cheque in Figure 3-12).

Although only the operationalising softgoals are reflected in the design and implementation (in the form of operations, data representation, assignments of tasks to an external agent, etc.), all softgoals participate in the construction of the SIG and can appear in supporting documentation for later development stages (e.g., the claim softgoal in Figure 3-12).

The level of offsprings' contribution can vary, depending on the type of contribution. If there is only one offspring, it can fully satisfy or deny the parent (in terms of NFR framework: make or break it); but if more than one offspring contributes to the parent, then their interrelationship counts (e.g., in case of AND relationship all offsprings need to be satisfied to satisfy the parent, while in case of OR – satisfaction of one is sufficient), as does each offspring's contribution level. This level can vary from *break* (strong negative), through *hurt* (weak negative) and *unknown* to *help* (weak positive) and *make* (strong positive).

NFRF suggests that developed refinement methods and correlations between softgoals both should be catalogued for future reuse.

[12] Source of Figure: adapted from [10] Figure 7.4

3.3.2.1.3 NFRF Process

The process of NFR Framework application (as presented in [10]) starts with acquiring knowledge about the domain and the system to be constructed, its functional requirements, and particular types of NFR and associated development techniques. From this the developer can identify particular NFRs for the domain relevant to the particular system.

Once identified, the high-level non-functional requirements become the top level softgoals which are then decomposed using the NFRF softgoal type and refinement catalogues. Through this process abstract softgoals, such as security, get refined into more specific non-functional requirements, such as confidentiality, availability, etc. For each sub-softgoal suitable operationalisations are then identified, providing solution alternatives for the sub-goals.

The softgoals as well as their operationalisations often have to strive for conflicting aims. While decomposing and identifying operationalisations, the developer has to consider ambiguities, trade-offs, priorities and interdependencies among NFRs and operationalisations. At this stage use of correlation catalogues is beneficial. The developer then selects some operationalisations, recoding and justifying the design decisions. After completing the initial SIG, s/he should evaluate the impact of the decisions and the whole process or parts of it could be repeated until a satisfactory result is produced.

3.3.2.1.4 Identification and Treatment of Crosscutting Concerns with NFRF

NFRF underlines the importance of non-functional requirements, which are, by their nature, crosscutting. The approach does not clarify how exactly the non-functional requirements are identified, only suggesting that they should be obtained by gathering knowledge about the domain for which a system will be built. NFRF focuses on clarifying the meaning of non-functional requirements and providing alternatives for satisfying them to the highest possible level, considering the conflicts between them, their interrelationships (assisted by correlation catalogues), as well as the developer's preferences.

The NFR SIG encapsulates the treatment of each crosscutting non-functional requirement, containing details of its decomposition, mapping to decisions and choices for realisation at the design stage.

On the other hand, the approach considers neither identification, nor handling of crosscutting functional requirements.

3.3.2.2 Knowledge Acquisition in Automated Specification (KAOS)

The KAOS approach has emerged from application of ideas from the machine learning domain to requirements engineering [35]. KAOS views requirements analysis as two coordinated tasks: requirements acquisition and formal specification [22]. The requirements acquisition task is focused on structuring requirements into a preliminary system model (*requirements model)*, using a rich modelling language. The formal specification task is focused on refinement of (part of) the requirements model into more precise formalism suitable for formal proofs and prototype generation. In the flowing section we mainly consider the first task, and only briefly address the second as it is more relevant for the formal specification work within the AOSD-Europe Formal Methods Lab.

3.3.2.2.1 KAOS Method

The overall KAOS approach has three main components:

1. *Conceptual model* for acquiring requirements models and its supporting language. The conceptual model in turn has three levels: meta, domain, and instance;
2. A set of requirement elaboration strategies;
3. An automated assistant for guidance through the strategies.

The meta-model level in the conceptual model provides a set of constructs for representing general domain-independent abstractions and their relationships necessary for modelling a system. Examples of such constructs are Agent, Action, Entity, Relationship, etc., as represented in the meta-classes level in Figure 3-13.

The meta-level abstractions are concretised into sets of domain-specific concepts when a given domain modelling is undertaken. For instance, in Figure 3-13, a library domain is modelled with Agent being specialised as Borrower, Entity as BookCopy, etc. The domain-level concepts are linked through instances of the Relationship abstraction, such as Borrower *performs* checkout, etc.

Finally, at the *instance level* (called Token level in Figure 3-13) individual instantiations of the domain model can be produced.

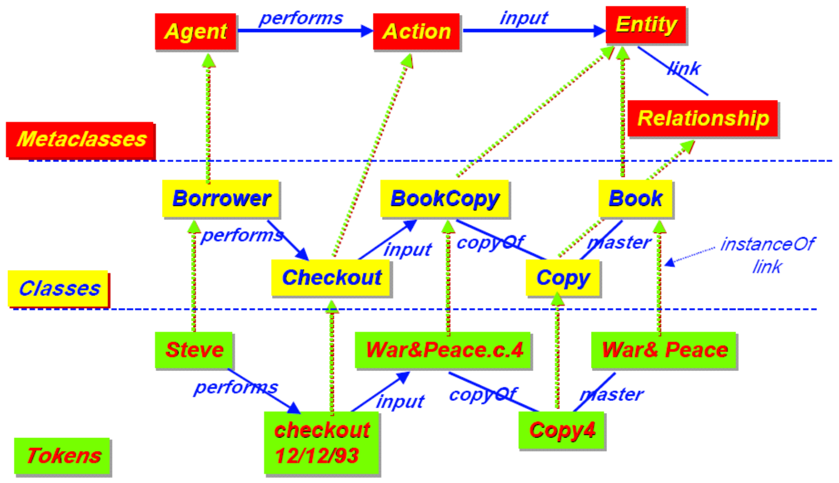


Figure 3-13 [13]: The Levels of KAOS Conceptual Model.

The conceptual meta-model in KAOS plays several central roles for requirements acquisition [22], to mention only a few, it suffices to point out that the structure of the requirements acquisition language is based on the meta-model abstractions and their relationships; each concept of the lower level inherits all properties of the corresponding concepts of the higher level; the meta-model drives the knowledge acquisition process. For instance, with a goal-directed acquisition strategy (explained below) the goal meta-concept is considered first, instances for it are acquired, for example, through *IsA* links, objects concerned with the goal are identified and so on, but all concepts of goals, links and objects are defined at meta-level initially.

The requirement elaboration *strategies* define specific ways of traversing the above described meta-model graph in order to acquire specific instances of various nodes and links. Strategies can differ by the meta-concepts around which they are centred: e.g., goal-directed, view-directed, and scenario-directed (i.e. centred around goals, views,

[13] Source of Figure [22] Figure 1.

and scenarios respectively) etc. The strategy is made up of finer steps such as answering questions, input validation against higher-level constraints, etc.

The *acquisition assistant* is used to guide the acquisition according to a strategy. It uses the requirements database and the requirements knowledge base. The Requirements database includes the requirements model built throughout the acquisition, and can be queried and analysed. The knowledge base includes the domain-level knowledge, such as concepts, hierarchies and fragments of requirements for a domain that can be reused and meta-level knowledge. The later relates to the properties of abstractions used in meta-model and can be represented as tactics for a strategy (e.g.,: the conflict between the goals can be temporarily tolerated, but will be resolved by some appropriate action [22]), etc.

○φ - φ is true in the next state
●φ - φ is true in the previous state
⏵⏵=xφ - φ will be true sometime (within x)
⏴⏴=xφ - φ was true sometime (within x)
□=xφ - φ will be always true (after some time)
■=xφ - φ was always true (until some time)
⌽φ U ψ - φ is true until ψ becomes true
⌽φ S ψ - φ has been true since ψ became true

Legend: circle – previous/next state; star – some time in the past/future state; square: always in the past/future

Figure 3-14: Representing Time in KAOS

For the formalisation of the acquired knowledge KAOS provides a formal language representation for all its abstract concepts as well as operators. For instance, Figure 3-14 demonstrates the time operators used in KAOS.

3.3.2.2.2 KAOS Artefacts

With KAOS multiple types of artefacts can be produced. For instance, one type is the knowledge structures (as demonstrated in Figure 3-13) produced for the domain. Instances of such structures are easily mapped to scenarios, thus resulting in scenario artefacts. They can also be formally specified using the KAOS formalisation language, and result in formal specification artefacts.

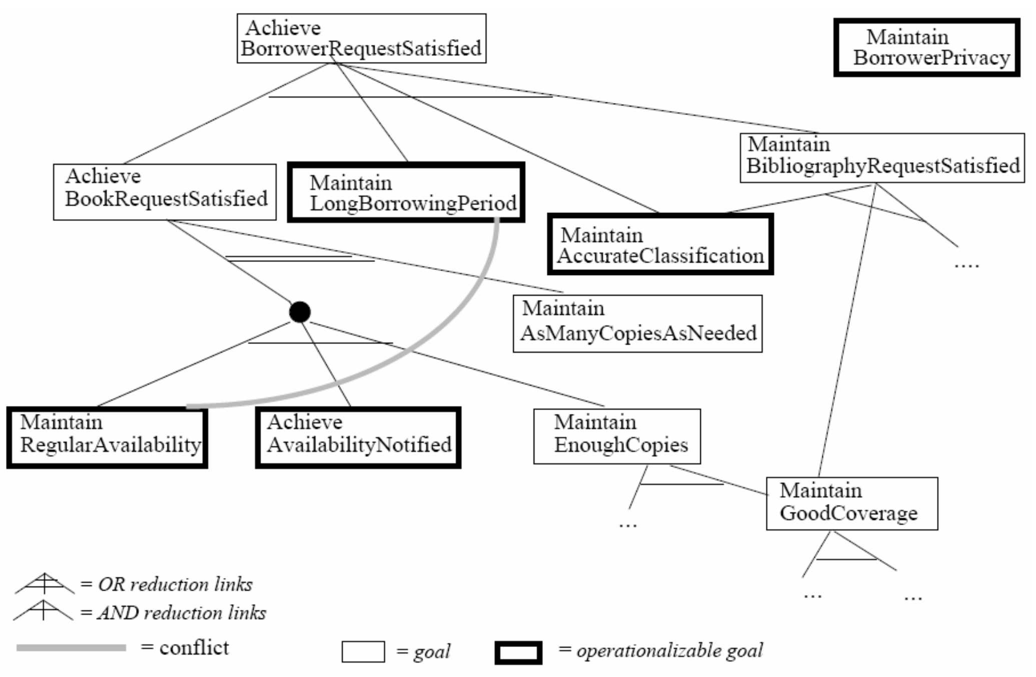

Achieve BorrowerRequestSatisfied
Maintain BorrowerPrivacy
Maintain BibliographyRequestSatisfied
Achieve BookRequestSatisfied
Maintain LongBorrowingPeriod
Maintain AccurateClassification
....
Maintain AsManyCopiesAsNeeded
Maintain RegularAvailability
Achieve AvailabilityNotified
Maintain EnoughCopies
Maintain GoodCoverage
...
... ...
= OR reduction links
= AND reduction links
= conflict
= goal
= operationalizable goal

Figure 3-15[14] **: Portion of Goals Structure for Borrower Goals**

On the other hand, the strategies and tactics derived for knowledge structure traversals are also artefacts which can also be represented formally or expressed as guidelines.

Yet, the most popular use of KAOS is for goal-directed strategy, which results in goal decomposition trees. An example of such a decomposition tree is presented in Figure 3-15 where the goal of satisfying a borrowing request from a library is decomposed into a set of related goals. Such decompositions can be facilitated via reuse of *generic goals* and their *reductions* structures collected in the knowledge base. For example, the decomposition of goal *BookRequestSatisfied* (part of Figure 3-15) can be specialised from the generic structure presented in Figure 3-16.

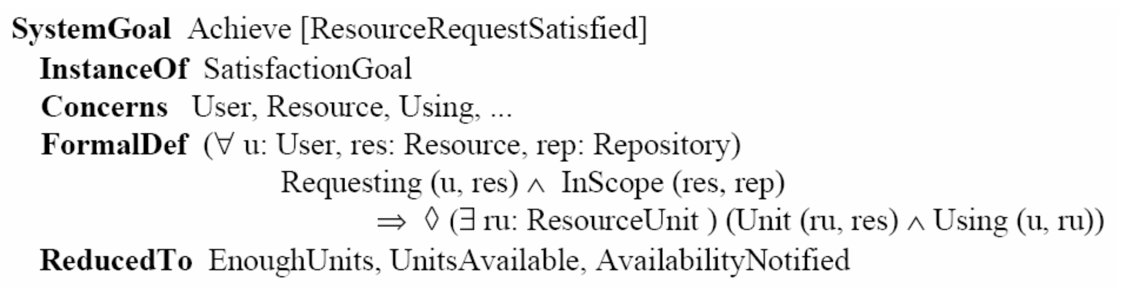

SystemGoal Achieve [ResourceRequestSatisfied]
  InstanceOf SatisfactionGoal
  Concerns User, Resource, Using, ...
  FormalDef ($\forall$ u: User, res: Resource, rep: Repository)
    Requesting (u, res) $\wedge$ InScope (res, rep)
      $\Rightarrow \Diamond$ ($\exists$ ru: ResourceUnit) (Unit (ru, res) $\wedge$ Using (u, ru))
  ReducedTo EnoughUnits, UnitsAvailable, AvailabilityNotified

Figure 3-16 [15]**: Requirement Fragment structure for KAOS knowledge reuse.**

The selection of structures for relevant decomposition is assisted through indexing schemes for retrieval based on goal *category* (e.g., satisfaction goal), *pattern* (e.g., achieve pattern), *links* between objects and goals (e.g., IsA).

[14]Source of Figure: [22] Figure 3.
[15] Source of Figure: [22]. Created from text on page 23.

3.3.2.2.3 KAOS Process

The KAOS acquisition process is largely defined by the selected acquisition strategy. For the case of Goal-Directed Strategy it can be reduced to the following steps [22] (which may overlap and iterate):

1. *Identify Goals and their Concerned Objects:* At this step main system goals are identified and categorised. The objects related to the goal are identified with their domain specific attributes and invariants, etc. Goals are decomposed and possible conflicts between them are identified. This is performed by analysts in cooperation with clients. The knowledge base structures are also identified and reused (as discussed earlier). Alternative goal decompositions are considered in order to minimise costs and resolve conflicts. By the end of this step a decomposition is selected with as few conflicts as possible and goals are decomposed to the level where they can be operationalised.
2. *Identify potential Agents and their Capabilities:* This step overlaps with step 1, it is concerned with identifying human agents, physical devices and programs which are capable of performing actions (i.e. causing stage change) on objects from step 1. Agent actions and corresponding pre/post conditions for them are defined.
3. *Operationalise Goals (here goal operationalisations are called Constrains):* The leaf goals from step 1 are converted into system objectives formulated in terms of objects and actions available to some agents. Several possible alternatives can be considered out of which the best one is selected. Generic operationalisations could be available from the knowledge base which may assist in this step.
4. *Refine Objects and Actions:* During steps 2 and 3 new goals, actions, objects and agents may be identified. In such cases it is necessary to iterate the above steps in order to complete the missing details about the newly identified items.
5. *Derive strengthened Objects and Actions to Ensure Constraints:* At this step the operationalisations defined in step 3 are "strengthened", i.e. additional pre/post conditions may be defined for some actions, trigger conditions may be provided, etc. This is carried out by reviewing the formal expressions for the operationalisations, checking actions and expected and occurring stage transitions against each other, and so on.
6. *Identify alternative Responsibilities:* Here the agents identified in step 2 are assigned to operationalisations in accordance with the capabilities of agents (also derived in step 2). The assignments state which agent will be responsible for each operationalisation. Alterative assignments will normally be considered.
7. *Assign Actions to responsible Agents:* At this step the actions are finally allocated to specific agents, the assignments are chosen from among the alternatives developed in step 6. This allocation implies that a given agent is contractually responsible for a given task. This allocation aims to maximise reliability of task fulfilment, minimise agent overloading, etc.

3.3.2.2.4 Identification and Treatment of Crosscutting Concerns with KAOS

Unlike most other classical RE approaches, KAOS explicitly sets out to deal with both functional and non-functional [22] requirements. The goal representation and decomposition structures of the approach and the formalisation language are well suited to address both types of requirements. KAOS also helps to study and analyse the relationships between the goals, thus allowing to single out the goals which relate to many others (i.e. the crosscutting ones). The only respect in which KAOS falls somewhat short is in holistic treatment of the crosscutting gaols and requirements: while it may have tactics and heuristics for dealing with functional and non-functional concerns, similar heuristics and tactics for dealing with crosscutting concerns have not been explicitly defined.

3.3.2.3 I*

The I* [18-21] framework provides an *agent based* approach to requirements engineering. The term *agent-based* reflects that the approach is centred around the system stakeholders and their relationships. These relationships reflect how actors depend on each other for achieving their goals, carrying out some tasks, and acquiring resources [18]. The support for explicit modelling and analysis of multiple actor dependencies allows to introduce some social analysis into a system analysis and design framework. The purpose of such analysis is finding answers to the "why?" of requirements: why are the requirements of one kind and not the other. Such studies, grounded in social/organisational context and rationale, help not only to understand current requirements, but also to prepare for future changes.

The approach also provides a set of simple intuitive graphical representations for actor and dependency modelling which facilitates the dialogue with stakeholders during requirement elicitation, encouraging stakeholder involvement and feedback.

3.3.2.3.1 I Method:*

I* approach is centred around agent and agent-relationship modelling. Agents are characterised in terms of their relationships (dependencies) with other agents. An agent that depends on another agent for a goal, task, or resource is called *depender*, that goal/task/resource is called *dependum*, and the agent dependent upon is the *dependee.* These dependencies allow an agent to achieve goals that he/she could not have achieved alone. On the other hand, these dependencies make the depender vulnerable if the dependee does not deliver the dependum agreed upon by both sides.

I* consists of 2 major parts: Strategic Dependencies (SD) part and the Strategic Rational (SR) part. The SD part studies actors and their dependencies, thus encouraging a deeper understanding of the business process. It helps to understand what is at stake in case of each particular dependency, who and how will be affected if a dependency fails.

In this framework an agent is *intentional*, i.e. possesses the qualities of [19]:

- *Intentionality*: agents want to fulfil certain goals or commitments, i.e. they have intentions. The intentionality can also be attributed to an agent (e.g., non-human agents) externally, by the modeller, as this provides a useful way of analysing an agent. Thus, an agent can be thought of as a locality for intentionality [19, 21]. Agents can also relate to each other at intentional level.
- *Autonomy*: agents are independent (i.e. have free will) and can act towards their goals, thus, they are not completely knowable or controllable by the modeller. Yet, the behaviour of an agent can be partially characterised using his/her known intentions.
- *Sociality*: agents exist within the network of relationships with other agents and environment. These relationships are multi-lateral, can be conflicting, and restrict an agent's behaviour.

- *Identity and Boundaries*: define who the agents are and what their responsibilities are. Not only physical entities, but also abstract ones (e.g., teams, roles) can be agents. The boundaries of an agent can change when a responsibility is re-assigned from it to another agent.
- *Strategic Reflectivity*: agents can reflect upon their actions and operation and re-evaluate these.
- *Rational Self-interest:* agents strive to achieve their goals, yet an agent may choose to constrain its strategies (in its own interest) due to social dependencies. However the rationality is partial and bounded due to the partial knowability of an agent and the limited resources available to the modeller.

The SR part investigates different alternative configurations of actors and their dependencies. This allows to systematically explore the space of possible new process designs [36]. Each alternative may have different implications for the agents and the system as a whole.

*3.3.2.3.2 I*Artefacts*

The central construct in I* is that of the *intentional agent*, as described in the above sub-section. Consequently, the artefacts relate to agent and dependency representation.

In the Strategic Dependency diagram example (demonstrated in Figure 3-17) actors are represented as cycle with their names inside.

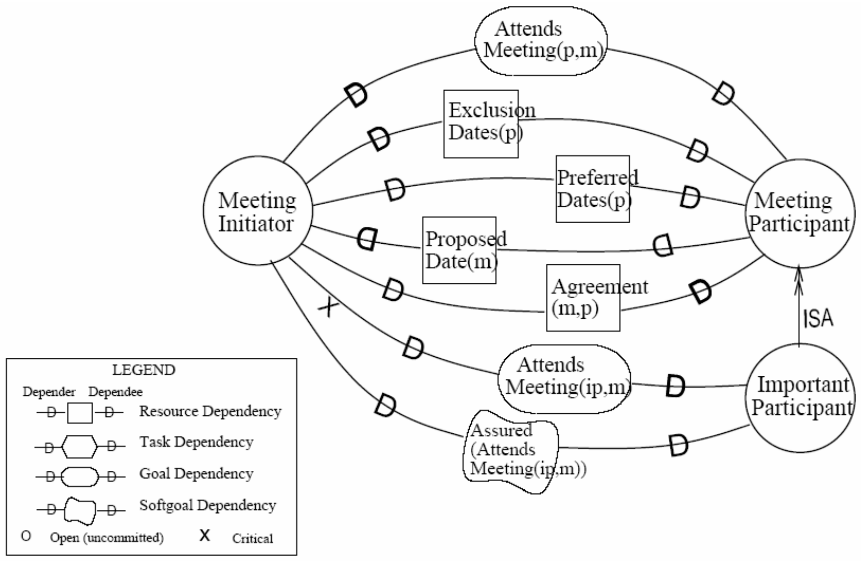


Figure 3-17 [16]: Strategic Dependency model for meeting scheduling without computer-based scheduler.

Besides the actors, the Strategic Dependency diagram includes four types of dependum: *goals, resources, softgoals,* and *tasks*, as well as the dependency links between agents. The *goals* are a kind of dependum which can be achieved in several alternative ways (e.g., attend meeting). *Softgoals* too are like goals, but either represent non-functional properties (e.g., assured to attend the meeting), or goals which are not quite clear at the time of representation. *Tasks* are dependum which should be achieved in a prescribed manner (e.g., attend meeting via video conferencing link). *Resources* are a dependum for which only the delivered result matters (e.g., proposed meeting date).

[16] Source of Figure: [18] Figure 1.

The links between the nodes in the SD diagram go from depender though dependum to dependee, and it is the depender who gets hurt if the dependum is not delivered upon.

The Strategic Rational Diagrams are used to further the study of agents and their dependencies by answer such questions as *why* (e.g., why does a certain even occur?), *how* (e.g., how does it occur?), *how else* (e.g., how else can the same event come about?). For instance, a meeting can be scheduled by an individual, as presented in Figure 3-18. This figure expands the agents presented in the Strategic Dependency diagram and looks “inside” of them, demonstrating how they agree upon the dependum and why each of them does so.

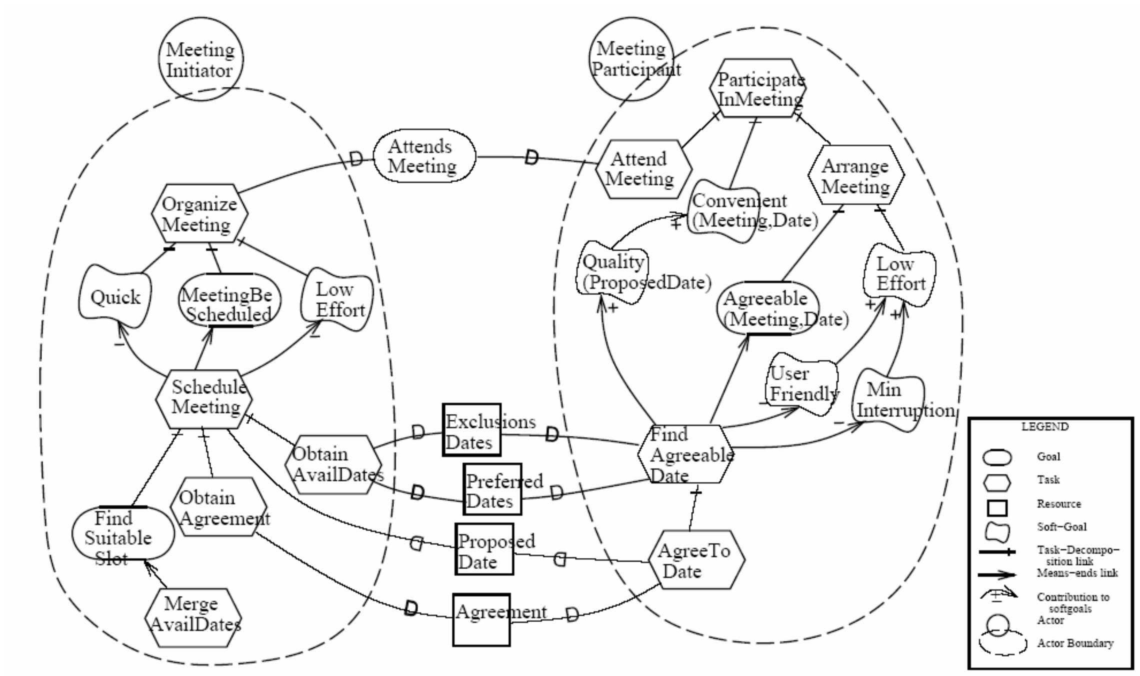


Figure 3-18 [17]: A Strategic Rationale model for meeting scheduler: manual scheduler

[17] Source of Figure: [18] Figure 3.

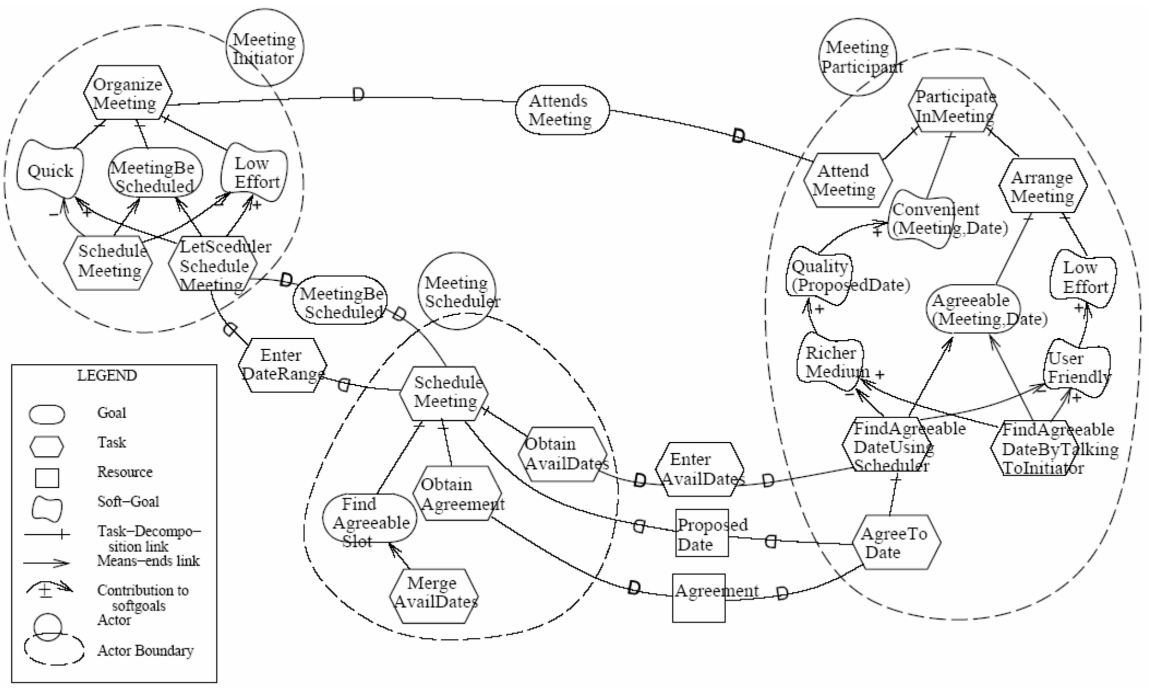


Figure 3-19 [18]: Strategic Rationale model for a computer-supported meeting scheduling.

Figure 3-19 demonstrates an alternative Strategic Rationale diagram for arranging a meeting: via a meeting planner software agent. This changes the boundaries of the Meeting Initiator agent by moving some tasks from him/her to the Meeting Scheduler agent.

*3.3.2.3.3 I*Process:*

The I* process begins by identifying the agents involved in a given process or system. The agents can be identified by considering who or what in the system has the *intentionality* characteristics discussed above. It is appropriate for the analyst to assign intentionality to non-human agents, if this assists with the analysis process.

Once agents are identified, the Strategic Dependency diagrams (Figure 3-17) are constructed for them, representing their top level goals and dependencies on other agents.

These SD diagrams are then elaborated into Strategic Rational diagrams by “looking inside” each agent. Here the structures of goal decomposition developed with NFR framework are used to analyse how an agent internally evaluates his/her goals and think about the procedures for their achievement.

By studying the SD and SR diagrams, the analyst may devise several alternative dependency structures and ways to achieve agents’ goals within these structures, i.e. alternative SD and SR diagrams. Each of these alternatives must be analysed for vulnerabilities: is a dependee motivated enough to deliver upon dependum? I* suggests that if there is a dependency loop between the depender and dependee (i.e. the dependee depends on the depender for another dependum either directly or via an intermediate agent), it is likely that the dependum will be delivered. Out of the analysed alternatives, the most suitable alternative can then be selected for realisation.

*3.3.2.3.4 Identification and Treatment of Crosscutting Concerns with I**

[18] Source of Figure: [18] Figure 4

As discussed above, the intentions of agents are manifested as high-level goals which can be either of a functional or non-functional nature. Often a behavioural goal may be complemented by qualifying non-functional ones [19]. The non-functional goals are handled similarly to NFR softgoals: through softgoal decomposition, while functional goals are decomposed into sub-goals and tasks. In both cases I* does not distinguish between crosscutting and non-crosscutting concerns.

3.3.2.4 Summary of Goal-Based Approaches

The representative goal-based approaches discussed in this section demonstrate the wide spectrum of usage of goals. The NFR framework focuses on the non-functional goals and their decomposition. I* underlines the importance of the *agents,* their characteristics, dependencies, and environment in goal analysis. Finally, KAOS is mostly looking at goal formalisation and knowledge acquisition. Thus, all three approaches are strongly complementary: NFR provides knowledge structures about the non-functional goals, I* provides the agent-related knowledge, and KAOS helps to formalise and reason about such knowledge. All three approaches use similar goal decomposition structure.

While the goal-decomposition structures of these approaches may serve as a good starting point for identification and treatment of crosscutting concerns, such explicit identification and treatment has not been applied in these approaches. Though there might well be, for instance, some guidelines on composition of some crosscutting goals in the KAOS domain knowledge base, the same guidelines may not be identifiable for other crosscutting goals. In summary, the issue of crosscutting has not been clearly addressed.

3.3.3 Problem Frames

3.3.3.1.1 Problem Frames Method:

The Problem Frames (PF) approach [23-26] proposes to deal with complex problems in software development decomposing them into structured sets of simpler interacting sub-problems with understandable interfaces. The correctly combined descriptions and solutions for the sub-problems then serve as the description and solution to the original problem. As stated in [23], PF addresses "... the topics that are often called functional requirements, software specification, and the path by which you get from one to the other".

The key point of the PF approach is decomposition of complex problems into *familiar* sub-problems, because if one has solved a similar problem in the past it will be easier for one to deal with the present problem. Thus, the approach sets out to identify the common simple problems which can be used as patterns onto which the complex problems should be decomposed. These classes of common problems are identified from the body of work on problem analysis for software development, in the same way as design patterns are identified from software design work.

The broad characteristics of these identified problem classes, along with descriptions of interaction of the real world problem domains with the intended computer system of the respective problem classes, are extracted and catalogued as *problem frames*. An example of such a problem frame is presented in Figure 3-20. This frame addresses the issue of building a software system that imposes certain behaviour on a part of the physical world.

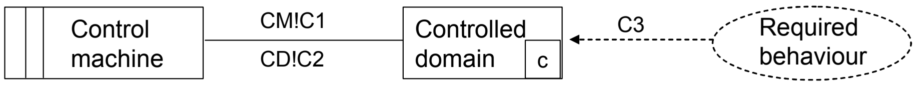


Figure 3-20 [19]: The Required Behaviour Problem Frame.

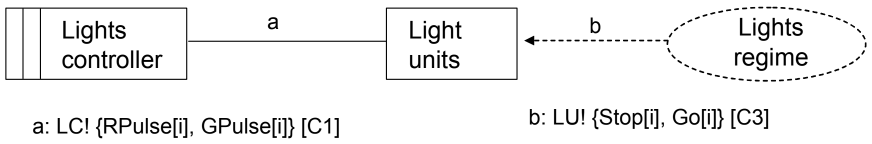
Several other problem frames, addressing specific problems are provided in [23].

3.3.3.1.2 Problem Frames Artefacts

The first artefact built in the Problem frames approach is the *context diagram* which simply represents the problem and the real world domains that interact with it.

As depicted in Figure 3-20, a *problem frame* itself is represented by a *problem diagram* consisting of the software part (generally called machine, but referred to as Control machine is the Required Behaviour frame), one or more domains (the *controlled domain* for this frame), the requirement (in this example it is required to control the physical world through software) and the shared phenomena between them.

An instantiation of the required behaviour frame (from Figure 3-20) is demonstrated in Figure 3-21 using the example of a one-way traffic light control. A light controller piece of software should control one-way traffic lights by sending red (RPulse) and green (GPulse) commands to the light unit; the requirement here is that Stop and Go signs should be demonstrated on the light unit for the red and green pulses respectively.



Lights controller
a
Light units
b
Lights regime
a: LC! {RPulse[i], GPulse[i]} [C1]
b: LU! {Stop[i], Go[i]} [C3]

Figure 3-21 [20]: A Required Behaviour Frame for Simple One-Way Traffic Light Control.

Each problem frame must have only one machine, which must address a distinct problem. Problem frames differ due to differences in their requirements, characteristics of problem domains[21], difference in involvement of the domains (is the domain affected, or monitored etc.) and the *frame concerns*. The *frame concern* of each problem class describes what kind of *descriptions* are necessary to adequately understand the given problem and what are the logical steps for its solution. A frame concern for the required behaviour frame for the example in Figure 3-21 is shown in Figure 3-22.

[19] Source of Figure: [23] section 4.3.1
[20] Source of Figure: [23] section 4.3.1.
[21] For instance, some domains have fixed casual interconnections, such as turning on/off the switch makes motor start/stop; others are less predictable, e.g., requesting a person to turn a switch may or may not result in switch being turned on. A classification of domains is provided in [23]

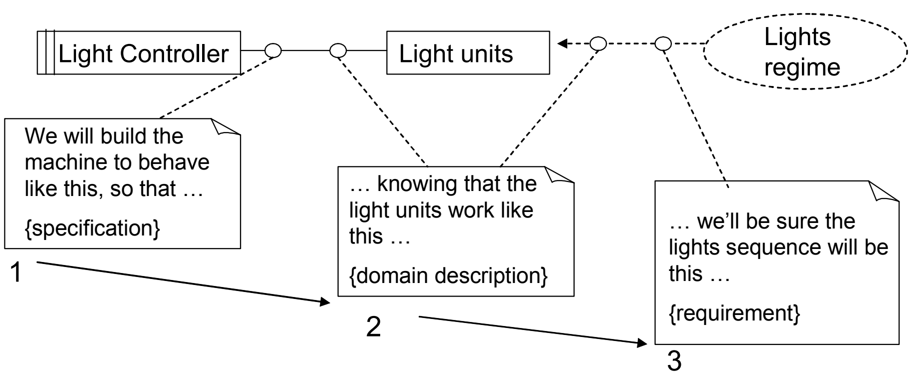

Light Controller
Light units
Lights regime
We will build the machine to behave like this, so that …
{specification}
… knowing that the light units work like this …
{domain description}
… we'll be sure the lights sequence will be this …
{requirement}
1
2
3

Figure 3-22 [22]: Frame Concern for Required Behaviour Frame.

From this figure we can see that a frame concern needs to have three complementary descriptions, named at the bottom of the notes at each step. The descriptions are for:

- Specification, which focuses on the computer system;
- Domains, which focuses on problem understanding;
- Requirement, which describes the problem from the customer's perspective.

Thus, the frame concern demonstrates that the specification is built to accommodate domain phenomena so that the requirement is met.

Since each problem class can have many variations, a number of variations and flavours (*variants*) are also considered for problem frames. A variant can add additional domains or features to the core problem in order to match the real problem at hand, but sill retains the core of the problem frame. Once the core problem frames for common classes of problems are known to the developers, it is easier to recognise and address the variations of these classes of problems and anticipate the difficulties as well as produce efficient solutions for them.

It must also be noted that since problem frames address simple sub-problems, a complex real world problem will normally be decomposed onto several problem frames which (along with their frame concerns) will need to be integrated later on to produce a complete problem and solution description.

3.3.3.1.3 Problem Frames Process

Development with PF starts with focusing on a problem, locating it in the real world and clarifying its boundaries by establishing what the software problem is and which real world domains it interacts with. This results in creation of the *context diagram* for the problem.

The context diagram is used to identify sub-problems and structure the problem as a collection of interacting sub-problems. The intention of this decomposition is to try and map the problem onto familiar simple sub-problems. During this activity, for each sub-problem a problem diagram is constructed. The diagram consists of relevant domains (selected from the domains of the context diagram), the projection of the machine (software) from the context diagram, and the sub-problem requirement.

Once sub-problems are identified, the developer should analyse each sub-problem and decide on the problem frame suitable for it. When the problem at hand fits a problem frame, it will satisfy, and so be solvable through, its frame concern. On the other hand,

[22] Source of Figure: [23] section 5.2.1.

if the problem frame for a given problem is selected inappropriately, it will be reflected by the frame concern: either the frame concern will require some descriptions which will not make sense for the problem, or some necessary descriptions for the problem will be unavailable in the frame concern.

However, in most cases the problem frame selected for a sub-problem will not fit perfectly: the problem is likely to be a variant or adaptation of the original frame. Thus, the problem frame should be adjusted correspondingly and the frame concern will need to be adapted to fit the sub-problem in hand.

After each sub-problem has a fitting problem frame, these should be composed to produce the *composite frame*, where *composition concern* is another issue to be addressed. Composition concerns between pairs of sub-problems arise mainly if they use different projections of the same problem domain. Here composition is considered in terms of:

- consistency between domain descriptions,
- precedence between inconsistent domain descriptions,
- interference between interactions with a domain, and
- scheduling of machines that interact with a common domain.

All these have to be addressed on a case-by-case basis for each individual problem. Finally, the frame concern of the composite frame should be checked to ensure that the initial problem is satisfactorily resolved.

3.3.3.1.4 Identification and Treatment of Crosscutting Concerns with Problem Frames

Although, as quoted above, the PF approach focuses on functional concerns, it also recognises the need to address non-functional ones. Yet, neither type of crosscutting concerns is identified or treated explicitly in any way. In [23] such crosscutting non-functional concerns as completeness, overrun etc. are described. These concerns, however, are not incorporated into problem frames, nor have a consistent treatment method. They are simply referred to as *other* concerns that are also important. The notable exception to this is the reliability concern, which is separated into a new problem frame and modelled in terms of countering the behaviour that the machine will display when interacting with the real world domains if failures occur. However, to the best of our knowledge, there is no sufficient work available to judge if this approach can be applicable to other crosscutting non-functional requirements.

3.3.3.2 Summary of Problem Frames

Problem frames provide a way of decomposing problems into a set of familiar sub-problems with known solutions. In their purpose and origin problem frames are similar to design patterns: they are intended to ease problem recognition and solution, and are derived from the "good practice" and experience of requirement engineers.

As discussed above, neither functional nor non-functional crosscutting concerns are identified and treated modularly in the Problem Frames approach.

3.3.4 Use Cases and Scenario Based Approaches

Scenarios and use cases have recently emerged as the most popular means of eliciting requirements in industry. A *scenario* is defined as "a sequence of actions carried out by intelligent agents" [27]. Scenarios (essentially short stories) capitalise on the innate ability of people to reason from stories. Thus when requirements are presented as scenarios, people find it easier to detect inconsistencies, omissions, and threats for the system to be built.

Scenarios with multiple options, conditions, and branches can be organised into *use cases*.

A comprehensive survey of the contemporary scenario-based work is presented in [27]. Most of the work described in [27] uses scenarios for requirement discovery, validation, and deriving tests from the scenarios. Here, we present the Use Cases approach [8] and a scenario-based approach for *misuse cases* (a variant of undesirable use cases) which can be used for addressing (a subset of) non-functional requirements.

3.3.4.1 Use Cases

3.3.4.1.1 Use Case Method

Each use case [8] describes a related set of system uses by some actors (i.e. end users of the system) to achieve some desired result. Use cases are used to systematically explore how a system is used by the stakeholders and highlight how different usage scenarios are handled in order to promote better understanding between users and developers and capture high-level, user-centric requirements.

In addition to helping to understand stakeholders requirements for the system's behaviour in a step-by-step fashion, this approach allows decomposition of a system onto components that "give measurable value to a particular actor" [37]. Thus, it also helps to derive the fundamental structure of the application as well as prioritise system functionality and facilitate incremental development, providing a use case as the smallest unit of delivery. Use cases also serve as a basis for early development of test cases and validation of acceptance criteria.

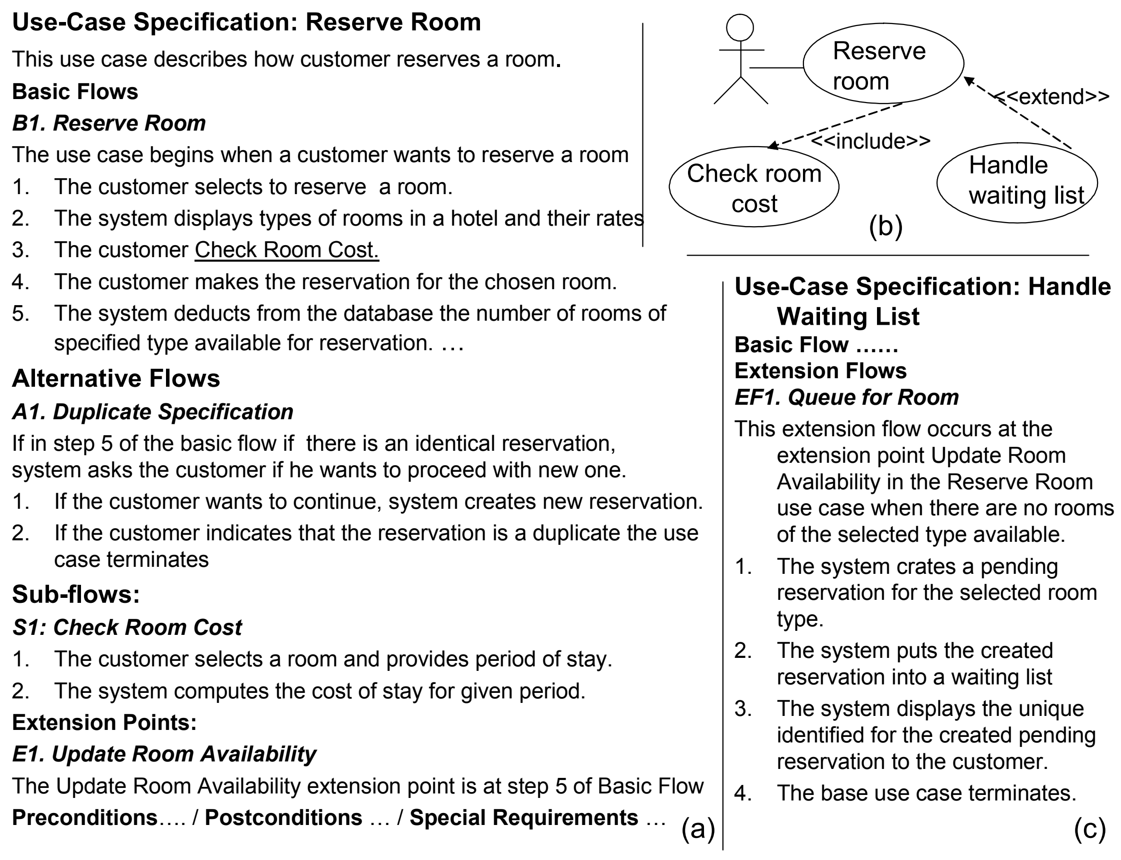
3.3.4.1.2 Use Cases Artefacts

The Use Cases approach distinguishes between *actors* ("what interacts with the system" [8]), *use case* ("what should be performed by the system" [8]), *use case instance* or *scenario* which demonstrates a particular instantiation of a use case, and *use case specification* which details the use case as a sequence of interactions between actors and the system. These artefacts are presented in Figure 3-23 .

Figure 3-23 (a) provides a detailed description of a use case for a room reservation in a hotel management system. An actor is represented as a matchstick figure. Each ellipse in Figure 3-23 (b) represents a use case that describes the functionality indicated by its name. Figure 3-23 (b) itself demonstrates the *use case model* which consists of the system stakeholders, its use cases and the relationships between these use cases. The relationships between use cases are modelled as *generalisation*, *extension*, and *inclusion*. *Inclusion* is used to factor out common behaviour between use cases (<<include>> stereotype in Figure 3-23 (b)). *Generalisation* is used to refine the sequence of actions of the more general use case through that of more specific ones (similar to the generalisation relationship of classes). Finally the *extend* relationship (<<extend>> stereotype in Figure 3-23 (b)) allows the addition of extra behaviour[23] to

[23] This does not have to be optional behaviour only. Required behaviour can also be provided as an extension use case in order to keep the base use case simple and clear.

the base use case, without having to change the base, where *base use case* encapsulates the core behaviour that is self-sufficient in providing "measurable value" to the users. The extra behaviour is encapsulated in an extension use case (Figure 3-23 (c)), which is inserted into the base use case at *extension points* (Figure 3-23 (a)). The extension points need to be pre-defined in the base use cases for every extension use case.



Use-Case Specification: Reserve Room

This use case describes how customer reserves a room.

Basic Flows

B1. Reserve Room

The use case begins when a customer wants to reserve a room

1. The customer selects to reserve a room.
2. The system displays types of rooms in a hotel and their rates
3. The customer Check Room Cost.
4. The customer makes the reservation for the chosen room.
5. The system deducts from the database the number of rooms of specified type available for reservation. …

Alternative Flows

A1. Duplicate Specification

If in step 5 of the basic flow if there is an identical reservation, system asks the customer if he wants to proceed with new one.

1. If the customer wants to continue, system creates new reservation.
2. If the customer indicates that the reservation is a duplicate the use case terminates

Sub-flows:

S1: Check Room Cost

1. The customer selects a room and provides period of stay.
2. The system computes the cost of stay for given period.

Extension Points:

E1. Update Room Availability

The Update Room Availability extension point is at step 5 of Basic Flow

Preconditions…. / **Postconditions** … / **Special Requirements** … (a)

Use-Case Specification: Handle Waiting List

Basic Flow ……

Extension Flows

EF1. Queue for Room

This extension flow occurs at the extension point Update Room Availability in the Reserve Room use case when there are no rooms of the selected type available.

1. The system crates a pending reservation for the selected room type.
2. The system puts the created reservation into a waiting list
3. The system displays the unique identified for the created pending reservation to the customer.
4. The base use case terminates.

(c)

Figure 3-23 [24]: (a) Use Case Description; (b) Use Case Model; (c) Extension Use Case Description.

As depicted above, the main use case (represented by the basic flow) can encounter several eventualities when delivering the required functionality. These eventualities are represented by alternative flows of the use case. A use case instantiation – scenario – can be used to illustrate each possible flow with a possible real example. A scenario can also be illustrated with an interaction or sequence diagram.

3.3.4.1.3 Use Case Process

The Use Case process starts by identifying the actors who will be using the system. This can be done, for instance, by talking to the stakeholders. Then, for each actor the set of required functionality from the system is collected. This required functionality becomes the set of use cases for that actor. All use cases together form the use case model.

Next, for each identified use case the most usual course of actor-system interaction is defined. This normally expected flow of interaction becomes the basic flow of the use case. The basic flow is described in the use case description (Basic flow in Figure 3.23(a)). If there is any additional functionality that complements the basic flow, it is

[24] Sources of figures [38]: (a) adapted form Listing 5-1; (b) used from Listing 6-1; (c) adapted form Figure 6-1

recorded as an extension use case, with its description completed as well (Extension flow in Figure 3-23 (a) and Figure 3-23 (c)). This should be a high-level description, not concerned with internal workings of the system, or user interface, etc. The basic flow description is followed by identification and description of alternate flows (Alternative flow in Figure 3-23(a)). Identification of flows can be facilitated through use of scenarios.

Finally the description of each use case is reviewed against the descriptions of the other use cases. The interaction flows common to more than one use case are separated into included use cases, and basic and alternative flows are also reviewed.

Further processing of the use case approach, related to use case realisations is considered a matter of architecture and design stages of software development lifecycle and is not reviewed in this section.

3.3.4.1.4 Identification and Treatment of Crosscutting Concerns with Use Cases

Use Cases are very useful in identifying high-level user-centric functionality and structuring that functionality into basic, extending and included categories. Thus, crosscutting functionality that affects more than one use case is likely to be factored out into included functionality, while secondary/complementary crosscutting functionality may be factored out into extension use cases. Thus, the user level crosscutting functionality is separated from base use cases. However, because use cases only address user-level functionality, crosscutting functional concerns related to the internal system functionality are not detected.

The non-functional requirements are not addressed by the classical use cases approach at all. However, there is more recent work on Aspect-Oriented Software development with use cases [38] that (among other things) attempts to address the identification and treatment of non-functional requirements. This is discussed in section 3.4.3.1.1.

3.3.4.2 Negative Scenarios and Misuse Cases

While the main body of work on use cases looks at ways of identifying the functionality to be provided by the system, some work has applied use cases for identifying what the system should prevent. Such negative use cases are called *misuse cases* [28-30], or *failure cases* [39], and a failure case is a special instance of a misuse case [29].

3.3.4.2.1 Negative Scenarios and Misuse Cases Method

“A misuse case is a negative form of a Use Case. It documents a negative scenario” [29]. The agents of misuse cases actively pursue either a hostile intent towards the system under development, or a goal which is damaging to the goals of the initial system. The actors in misuse cases could be humans with hostile intent, but also inanimate objects or phenomena which can metaphorically be “allocated hostile intent” [28]. Since the misuse cases reveal threats to the use cases, by documenting these one can identify means to *mitigate* such threats [28, 29]. Mitigation is often achieved by producing new use cases that provide new functionality.

The misuse cases method is particularly useful in security threat and safety hazard analysis. But it is suggested [28] that other non-functional properties, such as availability, maintainability, portability, etc. can also be documented via misuse cases. For instance, for maintainability the "hostile agent" could be an inflexible design or a wired-in device dependency, etc.

3.3.4.2.2 Negative Scenarios and Misuse Cases Artefacts

This method is essentially an extension of the use cases approach (section 3.3.4.1), consequently all the artefacts previously discussed for the use cases (i.e. actors, use cases, scenarios, use case specifications) are also applicable here.

The artefacts specific to this method are the *hostile roles* and the *misuse cases*. The hostile roles are those that want either to harm the system, the stakeholders, or their resources intentionally, or whose goals are incompatible with the system goals. These roles are documented as UML actors, however, an alternative notation to the stick figures can be used to represent the inanimate actors (e.g., weather in Figure 3-24 which shows an example use and misuse cases for a fur farming company which is threatened by animal rights protesters and rival companies.).

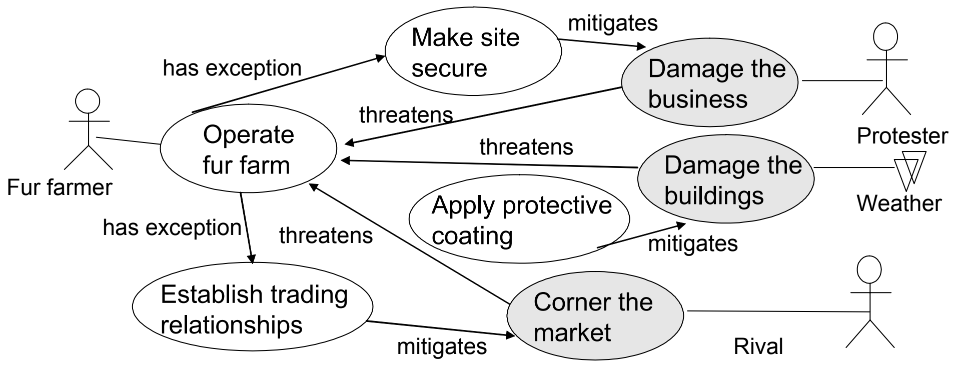


Figure 3-24 [25]: Misuse cases documented on the use case diagram.

The misuse cases need to be documented. In some simple cases the identified misuse case can be directly represented on the use cases diagram as misuse case handling or presenting requirement. For instance, a misuse case of an attack on one's property can lead to the need for the security requirement which will result in alarm subsystem to be represented as *install alarm* use case on the use case diagram. However, in more complex situations the misuse case is described as a bubble of inverted colour on the use case diagram (Figure 3-24) and then gradually analysed to identify how it can be handled or mitigated. A new *threatens* relationship is provided to illustrate which use cases a misuse case threatens. Another new relationship called *mitigates* is provided to demonstrate how the identified solutions could ease the affects of the misuse case. Finally a *has exception* relationship is used to demonstrate which use cases a solution to a misuse case helps to handle or prevent. Other stereotyped relationships, such as *conflicts with*, and *aggravates* can be used to demonstrate conflicts between use and misuse cases.

3.3.4.2.3 Negative Scenarios and Misuse Cases Process

The misuse cases process application is started in same way as that for the Use Cases (section 3.3.4.1), i.e. by identifying the agents and main use cases of the system. Then

[25] Source of figure: [29]. Adapted from Figure 7.5.

the method aims to identify the hostile agents and the goals that they might desire to achieve. Hostile roles can be identified in a workshop with the stakeholders.

Having identified the hostile agents (or alongside it), the misuse cases prompted by these agents can be identified. These misuse cases are then documented and the solutions to mitigate, or prevent them from being realised are developed.

For each identified threat at least one test case should also be developed to validate the proposed solution for the threat handling.

3.3.4.2.4 Identification and Treatment of Crosscutting Concerns with Negative Scenarios and Misuse Cases

As has been discussed for the Use Cases approach (section 3.3.4.1.4), use cases can be applied for identification and modularisation of crosscutting user-level functional concerns. This is also true for the misuse cases, as this approach is an extension of the Use Cases approach.

In the Use Cases approach agents are the people external to the system (or other external systems), which restricts the use cases to the user-level. The misuse cases approach, on the contrary, accepts that "hazards arise … from the system and subsystem functions" [29], and allows the subsystems to be specified as agents also. This, in turn, allows for the system-specific crosscutting functionality to be identified and modularised. It also implies that use cases can be defined for the design and implementation stages.

With regards to identification of non-functional concerns, the approach suggests to "think through scenarios to elicit constraints and non-functional requirements" [28]. Some non-functional requirements could surface from misuse scenarios, e.g., a hostile agent *software error* could "intend" to cause failure. The failure should be countered if the software is reliable. As a result reliability is identified as a requirement. However, this approach is more appropriate for understanding what the non-functional requirements imply, rather than their identification. For instance, thinking about scenarios that make the system fall short of reliability can demonstrate what the users expect from a reliable system.

3.3.4.3 Summary of Use Case and Scenario-Based Approaches

Today use cases and scenarios are the most widely used methodologies. We have discussed only two representative approaches from this space: the original Use Cases approach and its variation for misuse cases. The first one focuses on useful functionality of the system and system–actor interaction. The second one examines improper use of the system and its resources in order to develop solutions against such abuse.

Both the discussed approaches focus on treatment of functional concerns, largely disregarding non-functional ones. Although the Misuse Cases approach may address some non-functional concerns, this is more by an accident, rather than by design and such concerns are still largely disregarded. Besides, while use cases can modularly capture crosscutting functionality in extension and inclusion cases, there is no support for analysing, composing and resolving conflicts for such crosscutting concerns.

3.3.5 Summary of Non - AO Approaches

This concludes our discussion of a selected set of cotemporary Requirements Engineering approaches. While we had no ambition of presenting an exhaustive survey in this document, we have attempted to include the most influential pieces of work which have inspired the emergent work on Aspect-Oriented Requirements Engineering (discussed in section 3.4 below).

3.4 AO approaches

Requirements Engineering techniques that explicitly recognise the importance of clearly addressing both functional and non-functional *crosscutting concerns*, in addition to non-crosscutting ones, are called *Aspect-Oriented Requirements Engineering Approaches*. The emergence of these approaches is prompted by three major factors: need for composition, traceability, and development of new technology.

The first influencing factor is the realisation that integration of separated requirements artefacts (e.g., viewpoints, goals and softgoals, use cases, etc.) is a bottleneck in many requirement methodologies. For instance, it is difficult to integrate the viewpoints of individual stakeholders when producing the complete system, or it is hard to integrate the security and response time softgoals with the distributed data retrieval goals, etc. This difficulty is caused by scattering of a concern in many artefacts (e.g., security issues are present in many viewpoints in PREview) and intertwining of concerns (e.g., security and response time requirements may influence each other and constrain the data retrieval). This integration bottleneck makes it difficult to modularly view and reason about the concerns represented in several artefacts (e.g., the security concern in PREview is present in many viewpoints, but has no modular representation for itself). The inability to effectively reason about crosscutting concerns at the requirements leads to a poorer understanding of the stakeholders' requirements as well as their trade-offs.

The second factor is the need to trace crosscutting properties across the lifecycle of a software system. It is not just sufficient to identify and reason about crosscutting functional and non-functional concerns during requirements engineering. Once these concerns and their associated trade-offs have been established, it is essential that the software engineers can trace them to architecture, design, implementation and subsequent maintenance and evolution. Modularisation of crosscutting properties at the requirements level is the first step towards maintaining such traceability.

The third factor is the arrival of Aspect-Oriented Programming, which has put forward new concepts for modularising scattered and tangled (i.e. *crosscutting*) concerns and new composition mechanisms (such as *joinpoints* and *pointcuts*). Identifying and treating such crosscutting properties from early on in the software lifecycle helps ensure homogeneity in an aspect-oriented software development process and also contributes to the aspect traceability goal mentioned above.

Thus, Aspect-Oriented Requirements Engineering approaches focus on systematically and modularly treating, reasoning about, composing and subsequently tracing crosscutting functional and non-functional concerns via suitable abstraction, representation and composition mechanisms tailored to the requirements engineering domain.

In the following sub-sections we discuss the most prominent and well developed AORE approaches. The approaches are mainly grouped into similar broad classes, as the non-AO approaches discussed in section 3.3, except where relevant work on AO exists in for a class not identified earlier (e.g. a class for multi-dimensional RE work). The discussion on each representative approach within the group is structured in the same way as for the non-AO approaches: looking at the general method, artefacts, and the process. The selected groups and approaches are:

- viewpoint-based AO group represented by work on Aspect Oriented Requirements Engineering with Arcade [1, 40],
- gaol-oriented AO group represented by Aspects in Requirements Goal Models [41],
- use case- and scenario-based AO group represented by Aspect-Oriented Software Development with Use Cases [38, 42], Scenario Modelling with Aspects approach [43-45], and Aspectual Use Case Driven Approach [46, 47],
- multi-dimensional separation of concerns group [48] represented by Cosmos [49, 50] and Concern-Oriented Requirements Engineering [13, 51],
- AO component-based group represented by Aspect-Oriented Requirement Engineering for Component-Based Software Systems [52, 53],
- other approaches group represented by the Theme/Doc approach [54-56]

3.4.1 Viewpoint-Based Aspect-Oriented Approaches

Viewpoint-Based AO approaches extend the classical viewpoint work with the notions of crosscutting concerns and composition. Presently the only known viewpoint-based AORE approach (discussed below) is based on PRVeiw.

3.4.1.1 AORE with Arcade

The approach is called "aspect-oriented requirements engineering" as it is one of the first approaches to introduce aspect-oriented concepts at the requirements level. In fact [40] was the first paper to introduce the term "Early Aspects" which is now a defacto term for work on aspect-oriented requirements engineering and architecture design. In order to avoid confusion with the general topic of aspect-oriented requirements engineering (AORE) we will refer to this approach as "AORE with Arcade" from this point onwards, Arcade being the tool that supports the AORE approach in question.

3.4.1.1.1 AORE with Arcade Method

The Aspect-Oriented RE approach initially outlined in [40] and developed in [1] proposes a technique for separating aspectual and non-aspectual requirements, as well as their composition rules. This work has developed a general RE process model (demonstrated in Figure 3-27) which can be instantiated with any requirements engineering techniques.

In [1] a concrete instantiation of the model is presented, using PREview-like viewpoints and an XML-based composition mechanism [1]. In this instantiation, the aspectual requirements are similar to PREview *concerns* – broadly scoped requirements that crosscut user requirements derived from many viewpoints. However, in PREview, the aim is to discover requirements and produce a requirements specification document[26]. AORE with Arcade, in addition, aims to modularise and compose the requirements level concerns, not only producing a requirements specification document, but ensuring its

[26] This discussion is on use of PREview viewpoints for requirements specification, rather than process improvement.

consistency. This is achieved through detecting conflicts via requirement composition and handling the identified conflicts between viewpoints, concerns, and requirements.

The approach balances the need to formalise the concerns, viewpoints, requirements, and composition operators and rules to ease representation and proof generation with the need to keep the artefacts simple enough to be understandable for the end users (who will be involved in conflict resolution with these artefacts). Consequently, the approach uses XML for artefact representation, keeping them structured, semi-formalised, yet simple and understandable. An additional benefit of using XML-based composition rules and operators is the ease of their change and extension as required for an individual project.

Composition is supported via a set of composition operators, actions, and outcomes. An example composition rule is presented in Figure 3-26 in the section 3.4.1.1.2.

AORE with Arcade also assists in preparing the identified requirements for the next stage of the development lifecycle by providing directions for their mapping. The requirements document, and conflict resolution decisions produced via AORE with Arcade method are then provided as inputs to the PROBE framework [57, 58]. PROBE was developed to establish links between aspectual requirements and their later lifecycle-stage artefacts. It generates proof obligations for AORE artefacts and supports tracing these from requirements through architecture, design, and implementation stages.

The most important contributions of this approach are its ability to *separate and then compose* crosscutting and non-crosscutting requirements thorough simple yet powerful and flexible composition support; thorough conflict identification and resolution support, and the possibility of validating and tracing the requirements throughout the whole software development lifecycle.

3.4.1.1.2 AORE with Arcade Artefacts

The main concepts in the PREview-style instantiation of AORE with Arcade are viewpoints and concerns. An example of AORE viewpoint and concern is presented in Figure 3-25 which demonstrates part of a Portuguese toll collection system where a device (called gizmo) is credited at an ATM, installed in a car, and activated to pay tolls as cars pass the toll gate.

```
<?xml version="1.0" ?>
  <Viewpoint name="ATM">
      <Requirement id="1">
        The ATM sends the customer's card number,
          account number and  gizmo
         identifier to the system for activation
         and reactivation.
        <Requirement id="1.1">
             The ATM is notified if the activation or
           reactivation was successful or not.
           <Requirement id="1.1.1">In case of
             unsuccessful activation or
             reactivation the ATM is
             notified of the reasons for failure.
         </Requirement>
       </Requirement>
    </Requirement>
  </Viewpoint>
```
(a)

```
<?xml version="1.0" ?>
  <Concern name="Compatibility">
    <Requirement id="1">
     The system must be compatible
     with systems used to:
       <Requirement id="1.1">activate
        and reactivate gizmos;
       </Requirement>
       <Requirement id="1.2">deal with
         infraction incidents;
       </Requirement>
       <Requirement id="1.3">charge
         for usage.
        </Requirement>
    </Requirement>
  </Concern>
```
(b)

Figure 3-25[27]: Viewpoint and Concern Artefacts in AORE

[27] Source of Figure: [1] (a): Figure 2, (b): Figure 6.

In AORE with Arcade, both the viewpoints (Figure 3-25 (a)) and concerns (Figure 3-25 (b)) have requirements and sub-requirements (also referred to as *children* of the top level requirement) with their unique identification numbers. These identification numbers are used to refer to the specific requirements during composition, as shown in Figure 3-26(a).

The composition also uses a *Constraint* tag which defined how a viewpoint's requirements have to be constrained by a specific aspectual requirement. Constraint has actions, operators and outcome elements. While AORE provides a set of such reusable actions, operators, and outcomes, it also allows for introduction and use of new user-defined ones, as and when needed. Figure 3-26(b) provides some examples of these reusable elements and their semantics.

```
<?xml version="1.0" ?>
  <Composition>
    <Requirement aspect="Compatibility" id="1.1">
       <Constraint action="ensure" operator="with">
         <Requirement viewpoint="ATM" id="all" />
       </Constraint>
       <Outcome action="fullfilled" />
    </Requirement>
  </Composition>
```

(a)

Constraint Action:
Enforce: used to impose an additional condition over a set of viewpoint requirements.
Ensure: used to assert that a condition that should exist for a set of viewpoint
Constraint Operator:
During: describes the temporal interval during which a set of requirements is being satisfied.
With: Describes that a condition will hold for two sets of requirements with respect to each other.
Outcome Action:
Satisfied: used to assert that a set of viewpoint requirements will be satisfied after the constraints of an aspectual requirement have been applied.
Fulfilled: used to assert that the constraints of an aspectual requirement have been successfully

(b)

Figure 3-26 [28]: An example Composition and composition Actions, Operators, and Outcomes for AORE.

The *composition rule* defines the relationship between the *aspectual requirements* (i.e. requirements defined for the *concern*) and *viewpoint requirements* (the requirements of an individual viewpoint) at the granularity of an individual requirement. If one wants to include/exclude all the requirements of a viewpoint in a composition, it can be done by using *all* value for the *id* attribute of the *Requirement* element in the composition specification (e.g., Figure 3-26(a)). The *composition operators* of a Constraint tag define the type of relationship between the requirements, *actions* define how these relationships should be enacted; and outcomes define what should be expected as a result of the composition. Thus, the composition rule presented in Figure 3-26(a) states "**Compatibility Requirement 1.1** (that the system must be compatible with the systems used to activate and reactivate a gizmo: Figure 3-25(b)) must be **ensured with** regards to **all Requirements** of **ATM** with the **outcome** that the latter are **fulfilled**".

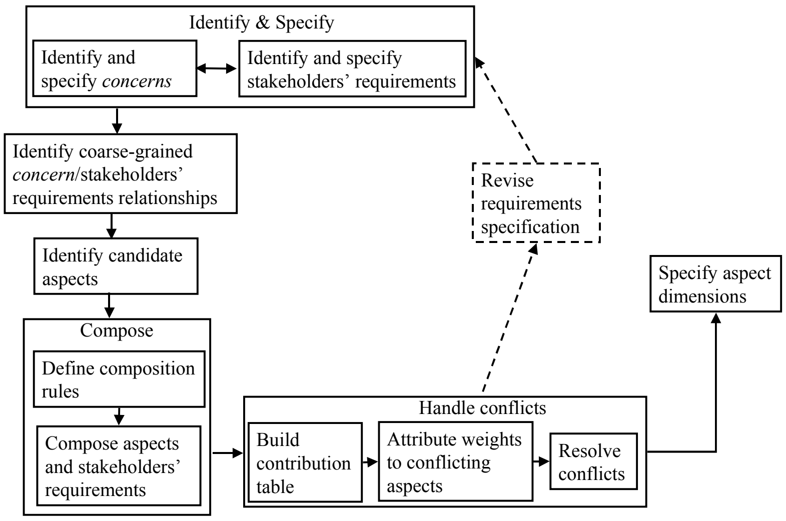
In addition to these XML-based elements, AORE with Arcade also uses matrices to relate concerns to each other and to viewpoints as well as to assign weights to conflicting requirements during conflict resolution.

Another important set of artefacts are the *proof obligations* generated by PROBE [57, 58] expressed in standard linear temporal logic.

[28] Source of Figure:
[1] Ibid. (a) adopted from Figure9; (b) adopted from Tables 4, 5, 6.

3.4.1.1.3 AORE with Arcade Process

The process of AORE with Arcade usage (as described in [1]) is depicted in Figure 3-27 below.



Identify & Specify
Identify and specify *concerns*
Identify and specify stakeholders' requirements
Identify coarse-grained *concern*/stakeholders' requirements relationships
Identify candidate aspects
Compose
Define composition rules
Compose aspects and stakeholders' requirements
Revise requirements specification
Specify aspect dimensions
Handle conflicts
Build contribution table
Attribute weights to conflicting aspects
Resolve conflicts

Figure 3-27 [29]: The Process Model for AORE

The process starts by identifying stakeholders' requirements. When using the PREview-based instantiation, this implies identification of viewpoints and collection of their related requirements. The identified requirements then are specified through provided XML templates.

The next step focuses on identification and specification of concerns. Unlike PREview, where concerns are discovered first and viewpoint requirements collected with these concerns in mind, AORE discovers concerns by analysing the initial requirements and also allows recursive concern and requirement identification. This is followed by identification of coarse-grained relationships between concerns and viewpoints. For this a matrix is built to mark which concerns affect which viewpoints. If a concern affects several viewpoints in the matrix it is considered a candidate aspect.

Next the developer defines composition rules between aspectual requirements[30] and viewpoint requirements at the level of an individual requirement [1]. At this stage the requirements engineer should specify how the identified viewpoint requirements and concerns should be composed and (if necessary) specify customised composition operators. Then the concern and viewpoint requirements are composed in accordance with the specified rules.

After composition the conflict handling is supported at two levels. First, due to the ability to compose aspectual and non-aspectual requirements at fine granularity, the need for analysing possible conflicts is eliminated when different concerns affect different requirements within the same viewpoint. Secondly, a mechanism for conflict assessment and resolution is provided when concerns do affect the same requirement. In this case a contribution matrix is to be built, with only negative contributions presenting a real problem. Then weight attribution based on fuzzy intervals is used to prioritise the conflicting requirements. Only if equal weights are assigned to conflicting requirements, there is a need for negotiation between stakeholders.

[29] Source of Figure: [1] Figure 1.
[30] These are called external requirements in PREview.

Finally, the process concludes by specifying *aspect dimensions*, i.e. deciding whether the aspect should be mapped to a *function, decision* or *aspect* at the later stages of software lifecycle [1].

All artefacts and documents produced via the above described process then form an input to the PROBE framework [57, 58], which gives temporal logic semantics to the natural language of the AORE-related terms, indicating what is to be proven about the implemented system with the given concerns, as opposed to a system without them. Additionally all UML design elements realising the AORE requirements should also be fed into PROBE, when available. Having treated all its inputs, PROBE produces proof obligations in temporal logic that should hold for the implemented aspects of the system. These proof obligations can be used both as input for formal model checkers or for test case generation to validate that the requirements have been correctly implemented.

3.4.1.1.4 Identification and Treatment of Crosscutting Concerns with AORE

This approach has a generic mechanism for concern handling which can easily suit both functional and non-functional concerns. However, it is not clear how the crosscutting functional concerns should be identified. Identification of crosscutting non-functional concerns is also not completely satisfactory, as there is no support (e.g., guidelines, tools, etc.) for the requirements engineer in detecting these from the requirements collected from the stakeholders.

On the other hand, the very recent work [59] on this approach turns to application of semantic linguistic analysis of natural language and requirements documents for aspect identification. Although this work is at an early stage at the time of preparation of this report, it already has a sound well structured mechanism for identifying both functional and non-functional crosscutting concerns. An aspect identification tool built on a corpus linguistics analysis tool WMATRIX [60] is presently under construction.

3.4.2 Goal Based Aspect Oriented Approaches

For the goal-based approaches it is common to have a goal or a softgoal contributing to several other goals. Such relationships clearly fit into the *crosscutting* pattern of *scattering* and *tangling*. However, presently there is not much work available for goal-based aspect-oriented requirements engineering.

3.4.2.1 Aspects in Requirements Goal Models (ARGM)

3.4.2.1.1 ARGM Model

The work presented in [41] argues that aspects can be identified during goal-oriented requirements analysis. Aspects are discovered from the relationships between functional and non-functional goals by decomposing these into sub-goals, sub-softgoals, and their operationalisations. When the goals and softgoals have been decomposed and operationalised, the correlation links between softgoals and related functional goals are established, as discussed in section 3.3.2.1. From the resulting Goal/Softgoal Interdependency Graphs the aspects can be detected as tasks with high fan-in (i.e. links to goals/softgoals to whose satisfaction a task in question contributes).

A specific type of graph, called V-graph, is used to represent the goal-softgoal-aspect relations.

3.4.2.1.2 *ARGM Artefacts*

This approach is based on goal and softgoal [10] approaches, thus all the artefacts for goals, softgoals, tasks, decompositions and alike used in those approaches are also used in here. In particular, in order to reason about the interdependencies between functional and non-functional requirements, V-graphs are used.

As shown in the Figure 3-28 (a), a V-graph consists of a functional goal, a non-functional softgoal and a task contributing to the satisfaction of both the goals.

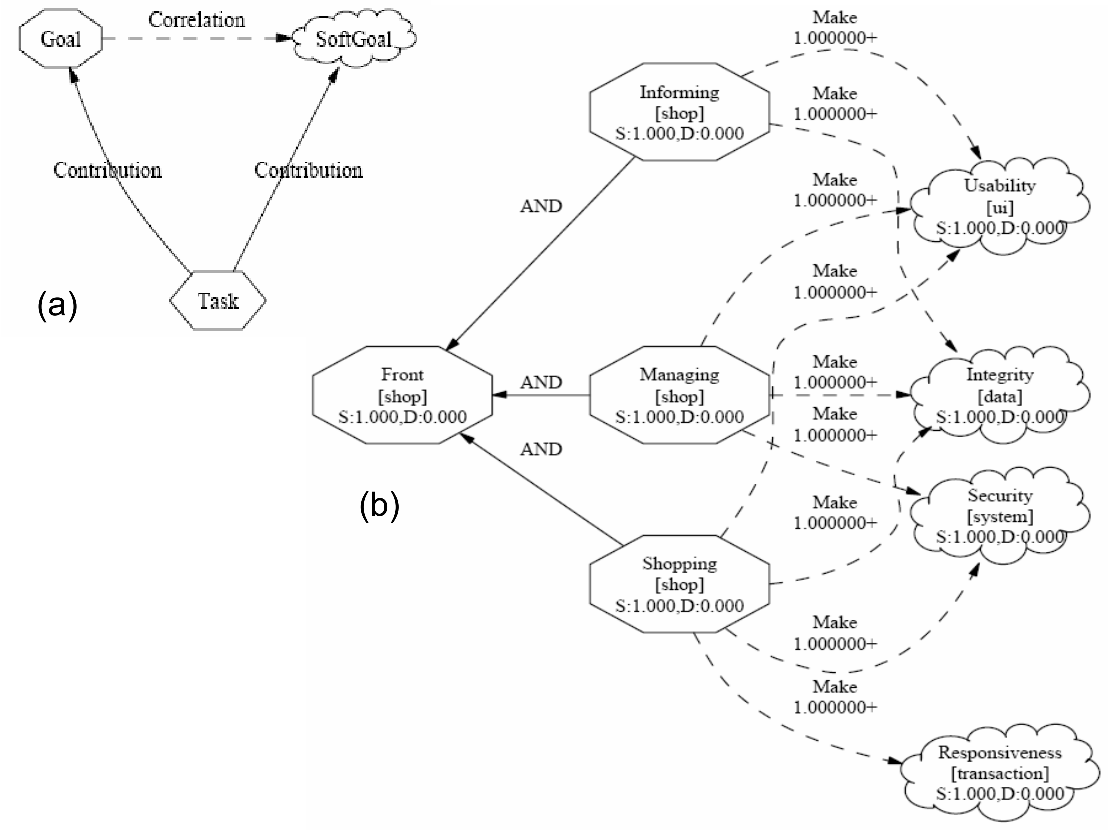

Goal
Correlation
SoftGoal
Contribution
Contribution
Task
(a)
Informing [shop] S:1.000,D:0.000
Managing [shop] S:1.000,D:0.000
Shopping [shop] S:1.000,D:0.000
Front [shop] S:1.000,D:0.000
AND
AND
AND
Make 1.000000+
Usability [ui] S:1.000,D:0.000
Integrity [data] S:1.000,D:0.000
Security [system] S:1.000,D:0.000
Responsiveness [transaction] S:1.000,D:0.000
(b)

Figure 3-28 [31]: (a) V-Graph Used to Link a Task of the Goal Hierarchy to a Softgoal as Its Operationalisation; Goals is an Octagon, Task – a Hexagon, a Softgoal – a Cloud. (b) Consistent Decomposition of Goals and Softgoals.

The use of V-graphs for decomposing goals and softgoals and generating the interdependency graphs is shown in Figure 3-28 (b) for an on-line shopping site example. On this interdependency graph, the contributions (S for satisfy, D for deny) and weights (ranging from 0 to 1) of sub-goals and sub-softgoals to their parents are also labelled.

After the goal and softgoal decomposition is complete, the goal interdependency graph is produced that contains both functional and non-functional goal-related tasks. Figure 3-29 shows part of such a graph for an on-line shopping site's shopping transaction (supported through a session cookie) which includes selecting the products, adding them to the cart, preparing to check out, and the actual checking out with its related tasks.

[31] Source figure: [41] (a) Figure 3; (b) Figure 9; (c) Figure 13.

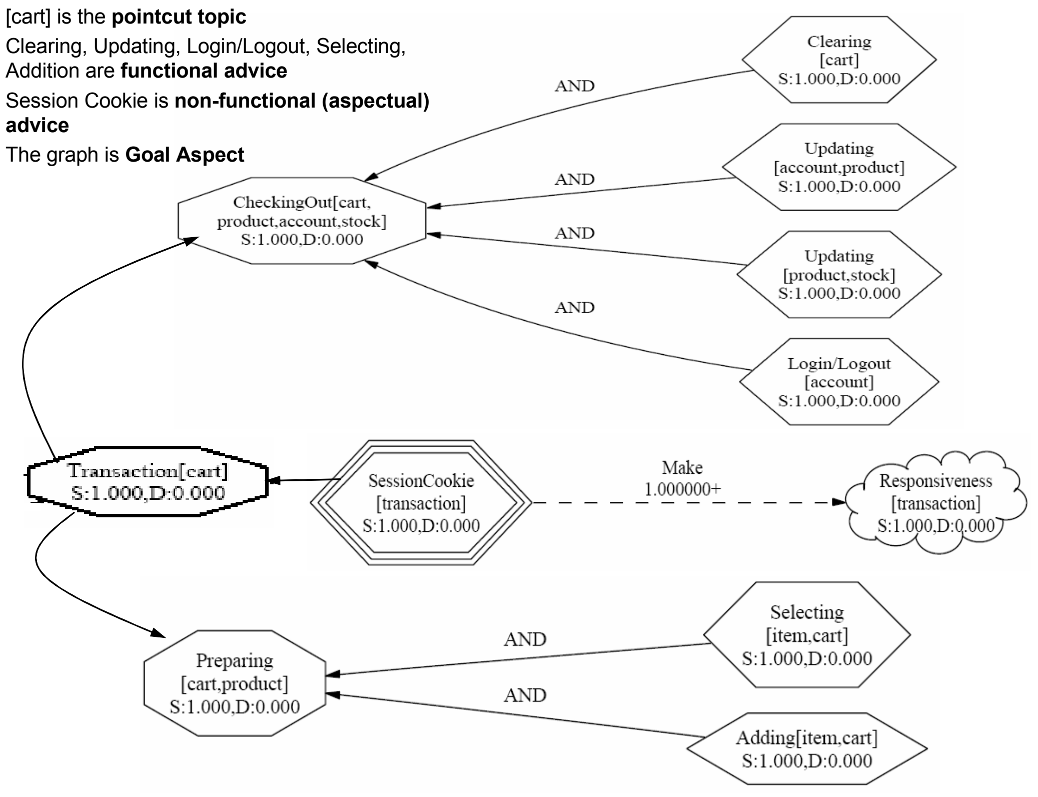

[cart] is the **pointcut topic**
Clearing, Updating, Login/Logout, Selecting, Addition are **functional advice**
Session Cookie is **non-functional (aspectual) advice**
The graph is **Goal Aspect**
Clearing [cart] S:1.000,D:0.000
Updating [account,product] S:1.000,D:0.000
Updating [product,stock] S:1.000,D:0.000
Login/Logout [account] S:1.000,D:0.000
CheckingOut[cart, product,account,stock] S:1.000,D:0.000
AND
Transaction[cart] S:1.000,D:0.000
SessionCookie [transaction] S:1.000,D:0.000
Make 1.000000+
Responsiveness [transaction] S:1.000,D:0.000
Preparing [cart,product] S:1.000,D:0.000
Selecting [item,cart] S:1.000,D:0.000
Adding[item,cart] S:1.000,D:0.000

Figure 3-29 [32]: Goal, Softgoal, and Task Graph

Similar to the NFR framework, each goal has a topic and a type. The type reflects the generic functional/non-functional requirement (e.g., in Figure 3-29 the goal is of *Transaction* type), while the topic captures the contextual information of the goal (e.g., in Figure 3-29 the *Transaction* is within the context of *cart* topic). In this approach topics are also used as pointcuts at which functional goals and tasks are related to non-functional softgoals (e.g., [cart] topic in Figure 3-29). The tasks are advising the goal with the given topic. Aspectual tasks are the operationalisation tasks for the non-functional requirements.

Additionally, the approach puts forward the concept of goal-aspect: the goal which aspectual task and functional tasks realise. This is illustrated in Figure 3-29. Also, it is suggested that the advising tasks can be separately viewed and worked on, for convenience, and later composed into the general SIG through their topics.

3.4.2.1.3 ARGM Process

This approach provides a set of clear processes for goal/ softgoal decomposition and, at the end, for aspect identification. A number of procedures and sub-procedures for this work are discussed in [41]. Below we consider the main procedures.

The decomposition begins with a procedure called *AspectFinder*, which takes as input a set of goal and softgoal nodes, iteratively breaks them into sub-goals and sub-softgoals (this is performed via user input) using the *Decompose* procedure, correlates these goals and their decompositions using the *Correlate* procedure, and resolves conflicts between goals/subgoals through the *Resolve Conflicts* procedure.

[32] Source of Figure: [41]. Adopted from Figure 14.

The *Correlate* procedure establishes an initial relationship between root functional goals and softgoals. The relationship is represented by one or more correlation links. The sub-goals propagate their contributions up to the parent goal satisfaction, over the correlation links.

The *Decompose* procedure is used to input new sub-goal nodes, defined from the parent goals and softgoals. If any of the sub-goals provides a negative contribution to the parent, the *Resolve Conflicts* procedure is invoked, which removes the link between the offending sub-goal and its parent goal. The refinement process must be monotonic: no goal/softgoal must become less satisfied due to decomposition.

The aspect identification procedure is called *ListAspects*. It gathers a set of tasks that contribute to a softgoal. Aspects are the operationalisations of the softgoals that contribute to the functional goals. It is suggested that aspects are named after the softgoals they operationalise.

3.4.2.1.4 Identification and Treatment of Crosscutting Concerns with ARGM

With ARGM the goals (functionality) and softgoals (non-functional concerns) are recursively decomposed until they can be reduced to a specific task. The crosscutting functional and non-functional concerns can then be identified as the tasks that contribute to several goals and softgoals. However, due to the decomposition mechanism which does not allow decomposition to sub-goals that negatively contribute to the parent goal, it is not evident that the user requirements are adequately mapped to sub-goals and tasks. For instance, a worst case scenario for such decomposition is illustrated in Figure 3-30.

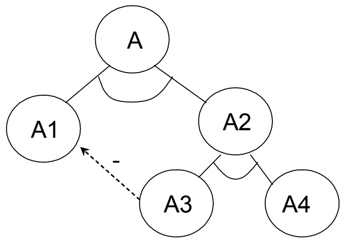


Figure 3-30: A worst case scenario for *Decompose* procedure

Here A3 negatively contributes to A1, thus the *Decompose* procedure will remove A3, which will deny A2 (since A2 requires both A3 and A4), and consequently, A itself (which requires A1 and A2).

3.4.3 Use Cases and Scenario Based Aspect Oriented Approaches

The use cases and scenario based approaches have been somewhat ahead of other classical approaches with regards to composition (e.g., through *extend* and *include* relationships). However, the functionality-centred nature of these techniques has left the issues of non-functional concerns not effectively tackled by them. Consequently the AORE work in this area extends the available composition support and attends to covering the non-functional properties.

3.4.3.1 Aspect-Oriented Software Development with Use Cases

Aspect-Oriented Software Development with Use Cases (AOSD/UC) [38, 42] suggests that use cases are crosscutting concerns, since the realisation of each use case affects several classes. This approach adopts an implementation-language-like view on use cases as aspects. This view is strongly influenced by AspectJ [61] and HyeprJ [62] languages. AOSD/UC adopts AspectJ-style linguistic constructs (e.g., joinpoint and pointcut) and HyperJ-type decomposition modules (e.g., slices).

3.4.3.1.1 AOSD/UC Method

AOSD with use cases [38] extends the traditional use cases approach [8] (discussed in section 3.3.4) with two main elements: *pointcuts* for use cases and grouping of development artefacts in *use case slices* and *use case modules*:

- Pointcuts here are groupings of use case *joinpoints* represented by *extension points* [42] and such elements as classes, operations, etc. within a use case. Whereas *extension points* (named points in the body of a use case specification) were available in use case modelling before AOSD/UC, these were only used to refer to a particular extension use case. In AOSD/UC extension points become a part of an 'extension point and pointcut' pair that helps to decouple the extending and base use cases and facilitates pointcut identification at the design stage.
- A *use case slice* contains the details of a use case at a given development phase, e.g., requirements or design. A *use case module*, on the other hand, contains all the details related to a use case, across all development phases.

The approach distinguishes two types of use cases: *peer* and *extension*. Peer use cases are distinct and independent of each other, each can be used separately with no reference to the other; they are the base requirements. When peer use cases are composed, their full operations are composed without intervention in their execution. However, composition of peer use cases needs to take into consideration the overlapping behaviour and conflicts between classes used for realisation of different use cases.

Extensions are additional features on top of the base use cases. Though extensions can be defined independently of the base cases, they should normally be used along with the base. When extensions are composed with the base cases, their operations usually interfere with the execution of the operations of the base use case.

The AOSD/UC also encourages the capture of non-functional requirements as use cases, using a construct called *infrastructure use case* which refers to the activities that the system infrastructure needs to perform to meet user requirements.

3.4.3.1.2 AOSD/UC Artefacts

All the artefacts discussed in section 4.3.4 for Use Cases are applicable in AOSD/UC, even if some will be slightly modified.

The UML use case representation as a classifier (Figure 3-31) is preferred to the ellipse representation because a classifier can provide more details about the use case. The classifier is extended with /basic/, /alt/, and /sub/ tags for basic, alternative and use case execution flows respectively. The *basic* tag indicates that the use case can be triggered by an actor, while the *sub* tag indicates that the flow can be referenced or included only by another flow. *Alternative* flows can be defined as extension use cases.

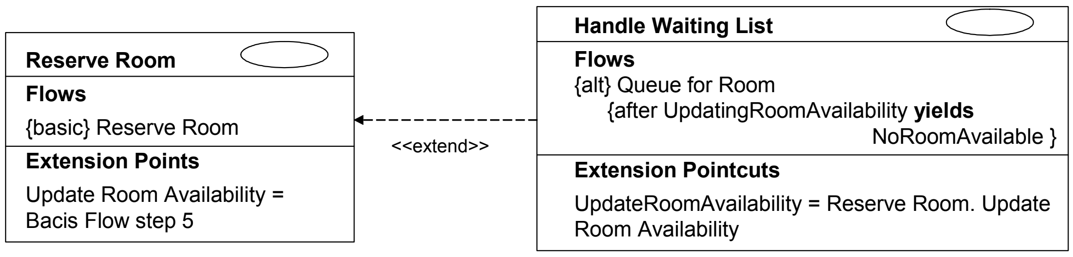

Reserve Room
Flows
{basic} Reserve Room
Extension Points
Update Room Availability = Bacis Flow step 5
<<extend>>
Handle Waiting List
Flows
{alt} Queue for Room
{after UpdatingRoomAvailability **yields** NoRoomAvailable }
Extension Pointcuts
UpdateRoomAvailability = Reserve Room. Update Room Availability

Figure 3-31 [33]: Use of UML Extended Classifier with Extension Points and Pointcuts for AOSD/UC, Using an On-Line Hotel Room Booking Example.

As demonstrated in Figure 3-31, the extension use case can be referenced from the base use cases through the names of these sub-flows (called *extension points*, provided in the *Extension Points* section of the use case representation). These extension points are in turn referenced from within the extension use cases themselves, via *extension pointcuts*. The flow definition in the extension use cases can also be complemented with an extension condition and *before*, *after*, *around* keywords, reflecting the conditions and sequence of extension point execution in the base use case flow. The extension points and pointcuts are later on used to develop design level pointcuts.

The newly introduced notions of *use case slice* and *use case module* in AOSD/UC are both used to localise use case related and complementary artefacts respectively at a specific lifecycle stage and across all stages. Some of the elements that can be contained in a requirements level use case slice are demonstrated in Figure 3-32.

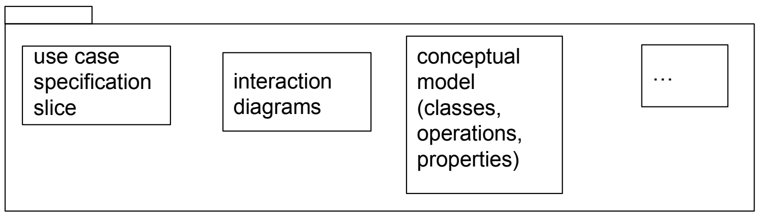

use case specification slice
interaction diagrams
conceptual model (classes, operations, properties)
...

Figure 3-32: Some Elements Contained in a Requirements Level Use Case Slice.

Similar use case slices should be produced for other stages of development, containing their corresponding elements. The design slice, for instance, should include the collaboration diagrams, classes, extensions, etc. for a given use case. All such slices are then packaged into a use case module, as shown in Figure 3-33.

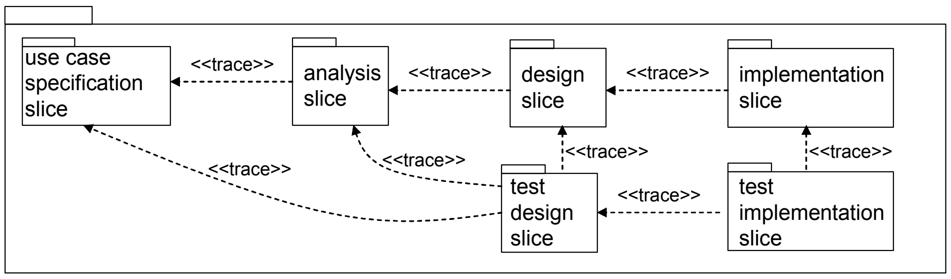
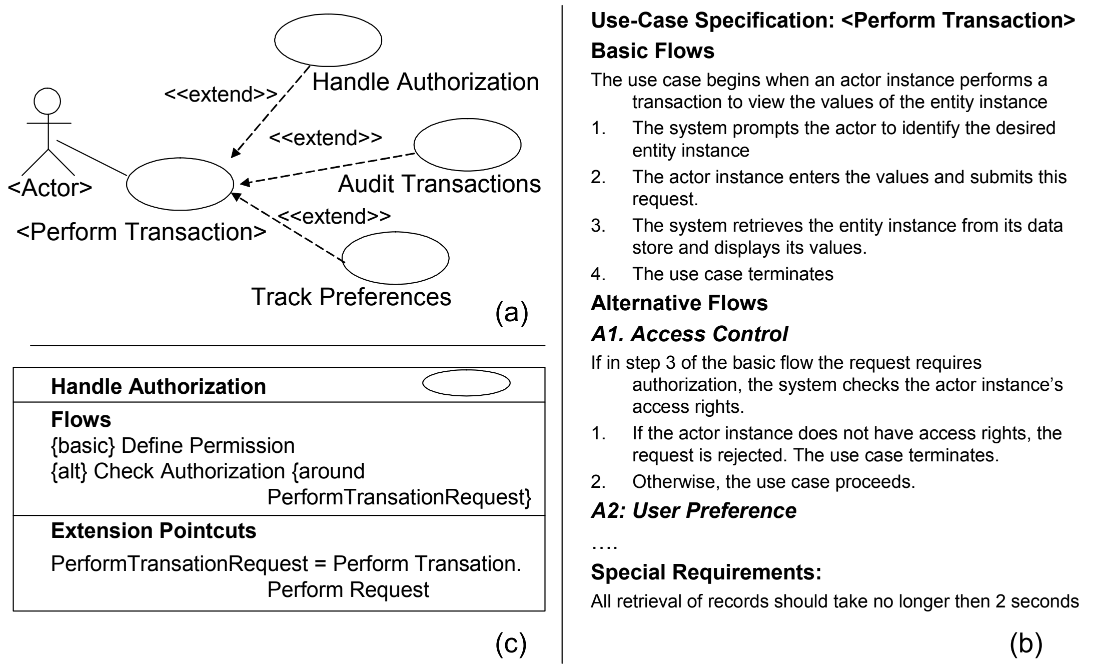


Figure 3-33 [34]: The Use Case Module.

[33] Source of figure: [38] Figure 6-4.

The special *infrastructure use case* <Perform Transaction> is used to capture the non-functional requirements. The template non-functional requirements should be represented as extension use cases to <Perform Transaction>, as demonstrated in Figure 3-34 below.



Use-Case Specification: <Perform Transaction>

Basic Flows

The use case begins when an actor instance performs a transaction to view the values of the entity instance

1. The system prompts the actor to identify the desired entity instance
2. The actor instance enters the values and submits this request.
3. The system retrieves the entity instance from its data store and displays its values.
4. The use case terminates

Alternative Flows

A1. Access Control

If in step 3 of the basic flow the request requires authorization, the system checks the actor instance's access rights.

1. If the actor instance does not have access rights, the request is rejected. The use case terminates.
2. Otherwise, the use case proceeds.

A2: User Preference

….

Special Requirements:

All retrieval of records should take no longer then 2 seconds

Handle Authorization

Flows

{basic} Define Permission
{alt} Check Authorization {around PerformTransationRequest}

Extension Pointcuts

PerformTransationRequest = Perform Transation. Perform Request

Figure 3-34 [35]**: An Example Infrastructure Use Case in AOSD/UC.**

The suggested advantages of using the template use case are that it helps to visualise the context of infrastructure mechanisms (such as authorisation, etc.) and serves as a base for systematically identifying extension points.

The approach also provides a number of design level artefacts, such as design level pointcut, parameterised use case slice, etc. These, however, are not considered relevant for this section and are not reviewed.

3.4.3.1.3 AOSD/UC Process

AOSD/UC is a development approach that spans the whole development lifecycle: from requirement gathering to implementation. Here, we focus on requirements engineering part of the process only.

For the requirements engineering part, the AOSD/UC process is very similar to that of the traditional Use Case process (discussed in section 3.3.4). The only significant difference is the inclusion of separate use cases for non-functional requirements. These should be identified, defined and recorded using the infrastructure use cases along with identification and processing of functional use cases. Another minor difference is the packaging of each use case and its related elements separately into a use case slice.

[34] Source of figure: [38] Figure 10.5.

[35] Source of figure: [38] (a) Figure 7.6; (b) adapted from Listing 7.3; (c) Figure 7.7.

3.4.3.1.4 Identification and Treatment of Crosscutting Concerns with AOSD/UC

Identification and treatment of crosscutting functional concerns by AOSD/UC at the requirements engineering stage is quite similar to that of the Use Case approach (discussed in section 3.3.4.1.4), with minor differences of using classifier representation with an Extension Pointcut section for use case representation.

In addition, AOSD/UC advocates applicability of use cases for non-functional requirement treatment: "so long as a requirement requires some observable response to be programmed into the system you can apply use cases" and "as long as you can define a test, you can define a use case for it" says [38]. As demonstrated in Figure 3-34, these are modelled as extensions to the Perform Transaction template use case.

However, the argument for the non-functional use case definition seems to disagree with the actual definition of a use case as *what the user does and what the system does in response*. A user does not *do* anything for the non-functional requirements, s/he simply requires them to be present. So there is no action involved from the user's side in most cases, for instance, when a user enquires about the weather over the internet, the user does not act with an intention to get a response within 2 seconds, or to synchronise his/her query with that of the weather update system – these are by-products of asking about the weather. Thus, the non-functional use case identification in this case should traverse in the following order: (a) what the user wants, (b) how it can be tested, (c) design test, (d) convert it into a use case, (e) extend the functionality with new use case.

Another uncertain area is that of aspect definition in AOSD/UC. As mentioned earlier, it is suggested that a use case is an aspect as it crosscuts many classes. This, however, assigns a characteristic to the requirements level concern on the basis of its design representation. Such a concern will certainly be crosscutting at the design level (with OO classes), but this might not be sufficient for it to be crosscutting at the requirements level as well.

3.4.3.2 Scenario Modelling with Aspects

Scenario Modelling with Aspects focuses on effects of aspects on behaviour modelling, taking a UML-based view. Here, the repeated behaviours is generalised into aspects, which are then instantiated when required.

3.4.3.2.1 Scenario Modelling with Aspects Model

In [43-45] an approach to modelling aspects as part of scenario-based modelling is proposed. The motivation for this approach is to assist in developing a more complete and consistent set of scenarios by removing the need to repeatedly deal with the same 'non-nominal, crosscutting' scenarios (e.g., failures, exceptions, etc.), which occur along with many scenarios.

The non-crosscutting and aspectual parts of scenarios are modelled separately from each other then merged as required, producing the complete scenarios. The composed complete scenarios can then be executed for validation purposes.

3.4.3.2.2 Scenario Modelling with Aspects Artefacts

This approach uses use cases for identification of functional requirements. Consequently, all artefacts discussed for the Use Cases approach (discussed in section 3.3.4) are also applicable here. An example use case diagram is presented in Figure 3-35

(a) for a parking lot use and payment example. Additionally, UML sequence diagrams are used for representing non-crosscutting scenarios (e.g., Figure 3-36(a)), from which Finite State Machines are generated using a state machine synthesis algorithm [63].

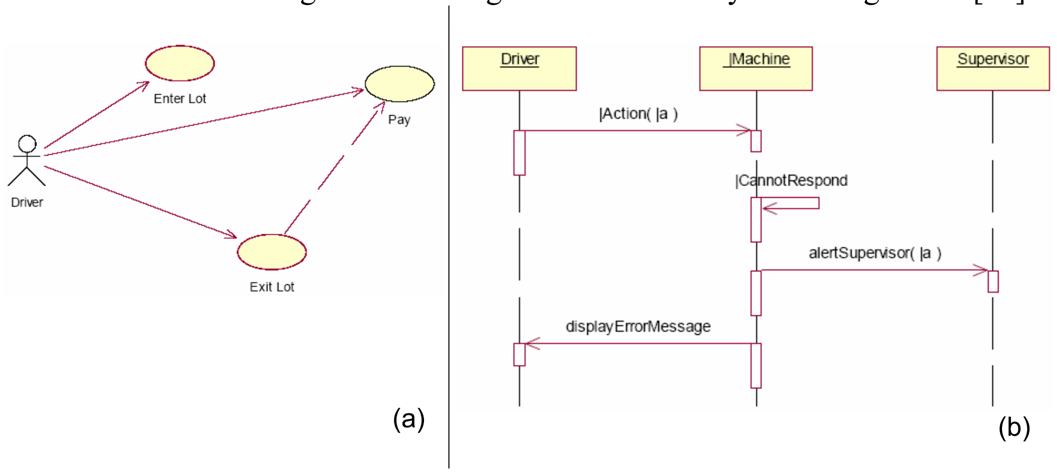


Figure 3-35 [36] **: Artefacts of the *Scenario Modelling with Aspects* approach: (a) An Example Use Case; (b) Interaction Pattern Specification Example.**

Aspectual scenarios are represented as Interaction Pattern Specifications (IPS) [64], where the IPS describes a pattern of interaction between its participants in terms of roles that the participants must fill in. It should be noted that this approach extends the notion of Interaction Patterns specification, allowing non-role elements to be included in it. An example of IPS is illustrated in Figure 3-35(b). While this might appear to be very similar to the UML interaction diagrams, some of the messages and their parameters have "|" prefixed to their names. This prefix indicates a role that needs to be filled by the participants instantiating the pattern.

The aspectual scenarios are translated from IPS to State Machine Pattern Specifications (SMPS) using a state machine synthesis algorithm [63]. An example for SMPS is provided in Figure 3-36(b). Again, the SMPS are quite similar to the UML state machines, but have role elements that need to be filled.

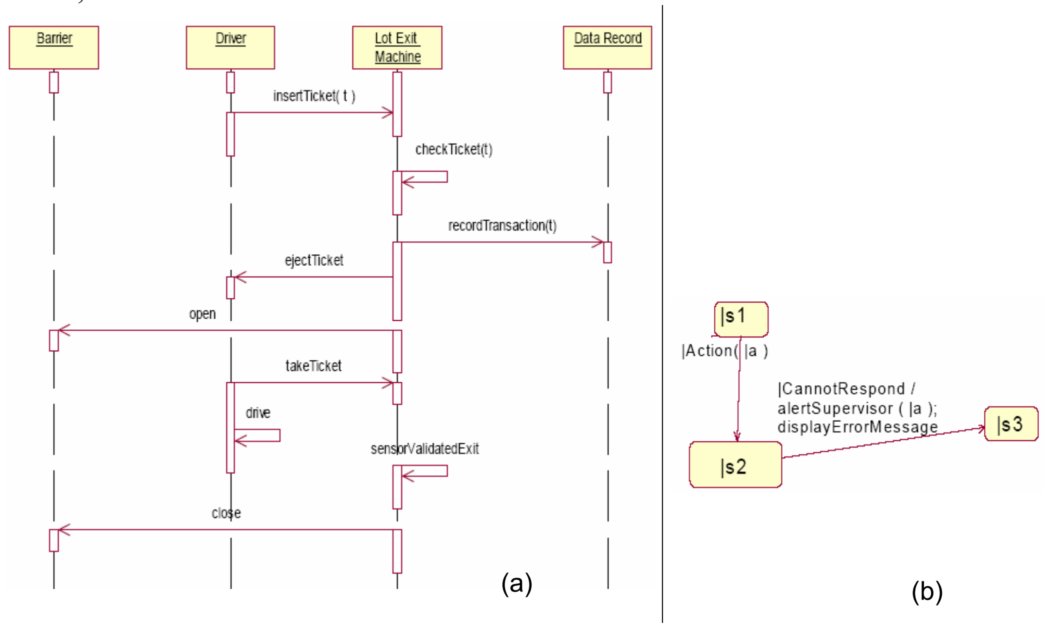


Figure 3-36 [37]**: (a) UML Sequence Diagram; (b) State Machine Patter Specification.**

[36] Source of Figures: [45] :(a) Figure 9, (b) Figure 10.

Before SMPS for aspectual scenarios and Finite State Machine (FMS) for non-aspectual scenarios are composed, the roles defined in SMPS need to be mapped to elements of the FMS. For this a State Machine Binding Specification is necessary. For instance, an example binding specification to bind the SMPS from Figure 3-36(b) to FSM of Figure 3-37(a) can be as follows:

|*s1* binds to *t1*
|*s2* binds to *t2*
|*Action* binds to *insertTicket*
|*a* binds to *t*
|*CannotRespond* binds to *timeout*

After this specification is provided, the state machines can be merged, producing the complete state machine presented in Figure 3-37b).

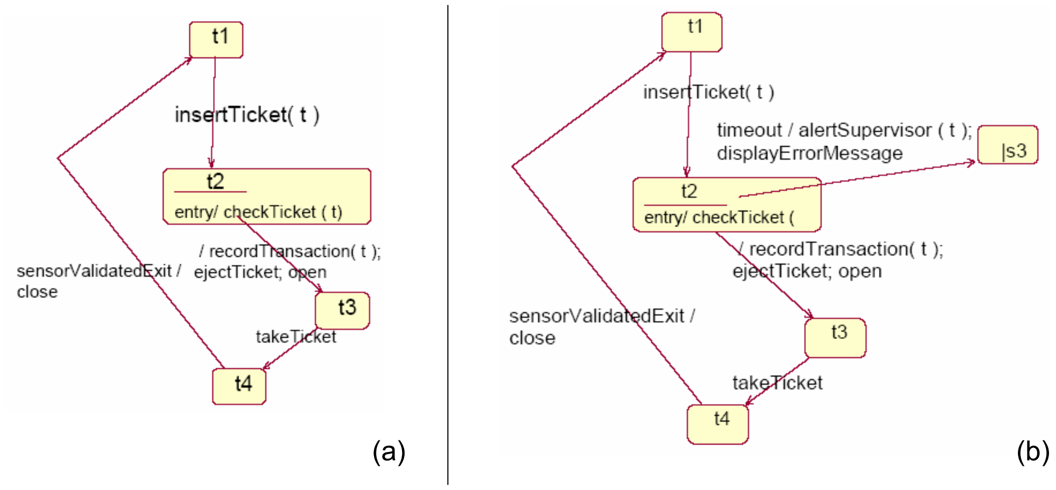

t1
insertTicket(t)
t2
entry/ checkTicket (t)
/ recordTransaction(t); ejectTicket; open
sensorValidatedExit / close
t3
takeTicket
t4
(a)
t1
insertTicket(t)
timeout / alertSupervisor (t); displayErrorMessage
|s3
t2
entry/ checkTicket (
/ recordTransaction(t); ejectTicket; open
sensorValidatedExit / close
t3
takeTicket
t4
(b)

Figure 3-37 [38] **: (a) Finite State Machine Produced form Interaction Diagram from Figure 3-36(a); (b) Composed FSM, Combined from State Machines on Figure 3-36(b) and Figure 3-37(a).**

3.4.3.2.3 Scenario Modelling with Aspects Process

Figure 3-38 presents the process of the *Scenario Modelling with Aspects* approach.

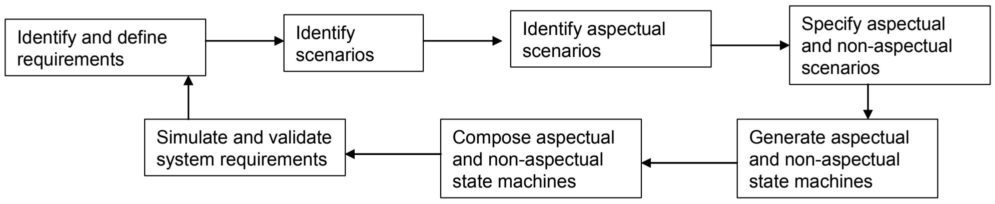

Identify and define requirements
Identify scenarios
Identify aspectual scenarios
Specify aspectual and non-aspectual scenarios
Generate aspectual and non-aspectual state machines
Compose aspectual and non-aspectual state machines
Simulate and validate system requirements

Figure 3-38 [39]**: The process of Scenario Modelling with Aspects Approach.**

The process commences with requirements identification and definition. [44] suggests that this is carried out using use cases to identify functional requirements and templates as in [46] for non-functional ones. This is followed by use case refinement and scenario identification. The non-functional requirements are likely to result in crosscutting scenarios, and other crosscutting scenarios may be identified for some functional requirements.

Source of Figure: [45] (a): Figure 11, (b) Figure 12.
[38] Source of Figure: [45] (a): Figure 13, (b) Figure 14.
[39] Source of Figure: [45] Figure 1.

Thereafter, the non-crosscutting scenarios are modelled as UML sequence diagrams (Figure 3-36(a)), then Finite State Machines are generated from them (Figure 3-37(a)) using a state machine synthesis algorithm [63].

Aspectual scenarios are modelled as Interaction Pattern Specifications (IPS; Figure 3-35(b)) [64], which are then translated into State Machine Pattern Specifications (Figure 3-36(b)) using a state machine synthesis algorithm [63].

The aspectual state machines are then instantiated and, finally, merged with non-aspectual ones using State Machine Binding Specification, which provides a correspondence between relevant states of each of the state machine.

Now, the composed state machine (Figure 3-37(b)) can be simulated using existing simulation tools. Alternatively, analysis tools for formal model checking techniques can be applied to prove conformance of the scenarios to the requirements.

When this process (discussed in [45]) is used the instantiation and merging of aspectual scenarios is carried out at the state machine level. An alternative process to this is discussed in [44] where the IPS for aspectual scenarios are instantiated, producing interaction diagrams; then the interaction diagrams for aspectual and non-aspectual scenarios are merged, followed by the generation of a composed state machine. When merging is carried out with interaction diagrams, the operators detailing how to interleave messages from different scenarios also need to be provided. Yet, both the processes lead to the same end and are both acceptable.

3.4.3.2.4 Identification and Treatment of Crosscutting Concerns with the Scenario Modelling with Aspects Process

The approach does not address the issues of aspectual scenario identification, merely stating that scenarios that represent use cases which affect other use cases (such as failure handling) are aspectual. The approach also suggests that the non-functional requirements will result in crosscutting scenarios. Such scenarios are illustrated in [44] as behaviours constraining the main functionality (e.g., preventing client connection while some processing is in progress). While this could indeed be a part of the non-functional requirement representation, further discussion is necessary to clarify how fully such scenarios can represent non-functional requirements. Nevertheless, once identified, the treatment of functional and aspectual scenarios is well represented and justified.

3.4.3.3 Aspectual Use Case Driven Approach

The Aspectual Use Case Driven Approach is similar to AOSD/UC in that it separates the crosscutting functionality into inclusion and extension use cases and also suggests to model quality attributes as use cases. However, here quality attributes do not have to be attached to an infrastructure use case. Also this approach provides a structured way – a template - for identification of crosscutting concerns.

3.4.3.3.1 Aspectual Use Case Driven Approach Method

The Aspectual Use Case Driven Approach [47] is concerned with extending the use case model to integrate non-functional requirements and identify the crosscutting functional use cases. The separated use cases for all concerns should also be integrated into a complete model using the provided composition mechanism.

This model extends the work on crosscutting quality attribute identification and integration into requirements engineering carried out in [46] where the crosscutting quality attributes are considered as fundamental modelling primitives; and it is further developed in [65] with support for unanticipated requirements change handling.

3.4.3.3.2 Aspectual Use Case Driven Approach Artefacts

Since the Aspectual Use Case Driven Approach is a use case based approach, all the artefacts discussed for the Use Cases approach (such as use case diagrams, etc.) are also appropriate and used for this approach.

Additionally, in this approach the quality attributes are initially represented with the template depicted in Figure 3-39. The template itself is influenced by [10, 66, 67]. In order to identify if the quality attribute is crosscutting, one needs to consider the information provided in the *Where* and *Requirements* rows of the template. If the quality attribute traverses several use cases/models and requirements, it is crosscutting.

Name	The name of the quality attribute
Description	Executive 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 the actors influenced by the quality attribute and also a list of models (e.g. use cases and sequence diagrams) requiring the quality attribute
Requirements	Requirements describing the quality attribute
Contribution	Represents how the quality attribute affects other quality attributes. This contribution can be positive (+) or negative (-)

Figure 3-39 [40]: Template for Quality Attributes.

The Aspectual Use Case Driven Approach also employs the notions of *overlapping*, *overriding* and *wrapping* of functional requirements by non-functional attributes. The *overlapping* of a functional requirement by an NFR implies that the NFR modifies the functional requirement (FR) by being applied *before* or *after* it. *Overriding* implies that the behaviour described by the quality attribute substitutes the behaviour of the FR. Finally the *wrapping* implies encapsulation of the FR by the NFR. Examples of such relationships are presented in Figure 3-40 and Figure 3-41 (using the Portuguese toll collection example mentioned in section 3.4.1.1.2).

Figure 3-40 displays how Response Time attribute is *wrapping* certain functionality of Gizmo, Vehicle, BankAccount and PriceTable classes. These classes will be involved in corresponding scenarios and use cases listed in the *Where* clause of the Response Time

[40] Source of figure: [46] Table 1.

NFR template. Their corresponding requirements will also be visible in the *Requirements* section of the template.

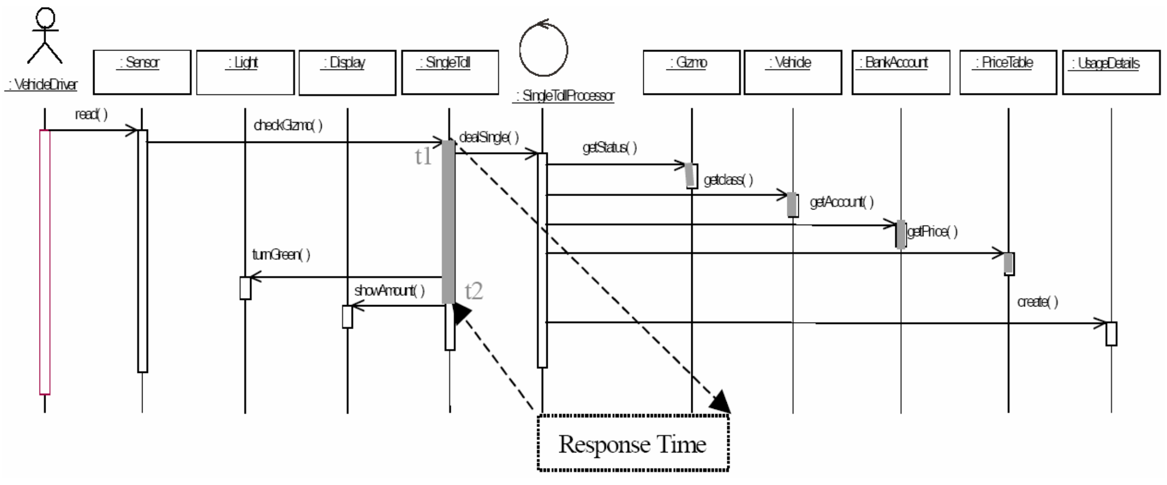


Figure 3-40 [41] : Response Time Quality Attribute Wraps the Functional Requirements.

Figure 3-41 displays the *overlapping* of Pay Bill and Register Vehicle use cases with the Security NFR. Here confidentiality must be used before the executions of the use cases and integrity must be used after them. Correspondingly, the dashed line represents *after* conditions while the dark line shows *before* conditions.

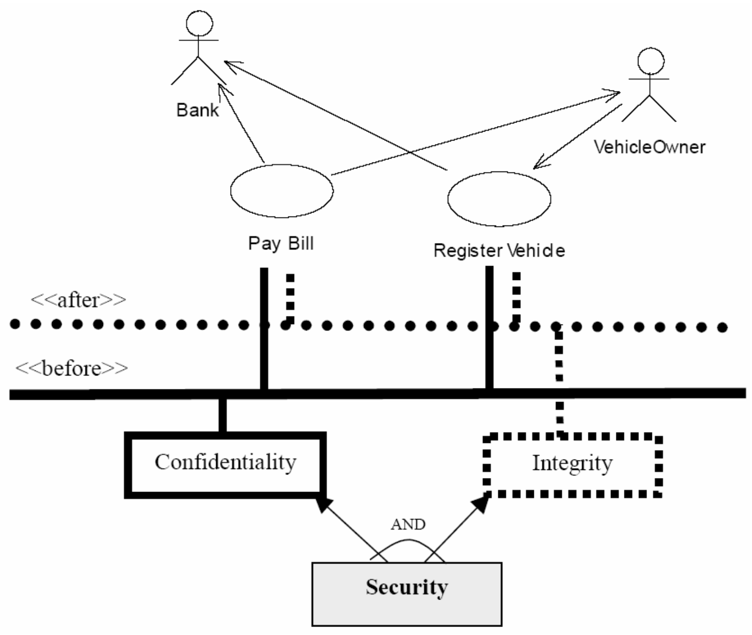


Figure 3-41 [42]: Security Overlaps Use Cases.

In [65] it is suggested to extend the set of relationships between use cases, complementing the standard UML *include, extend,* and *inherit* relationships with new ones for *collaborate, damage and constrain*. *Collaborate* reflects the positive contribution of one use case to the other, *damage* the negative contribution, and *constrain* reflects that a global property restricts another use case. An example of such use of *constrain* in use case diagrams is presented in the Figure 3-42.

[41] Source of figure: [46] Figure 5.
[42] Source of Figure: [46] Figure 6.

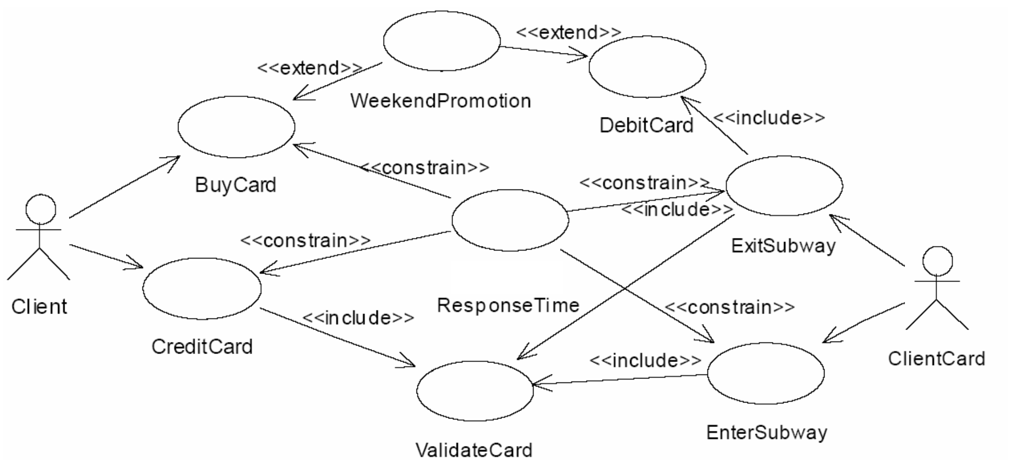


Figure 3-42 [43]: Projecting Response Time quality attribute on the use case diagram, using <<constrain>> relationship.

Another new artefact used in this work is the Use Case Pattern Specification (UCPS) [65] applied to represent the generic use cases in an abstract way which can later on be instantiated to suit a particular situation.

The idea of UCPS is based on the idea of Interaction Pattern Specification proposed in [68]. Use case roles represent the concerns that are more likely to change over time and these can be instantiated differently for particular configurations of the system. To describe these use cases in a generic way the generalised UML activity diagrams can be used. The generalisation of these diagrams is in inclusion of role elements into their definition, producing the Activity Pattern Specification (APS), in the same way as for UCPS. The APS are instantiated during use case composition, with assignment of role elements and a chosen composition operator. Such a composition example is presented in Figure 3-43.

```
Compose <use case A> with <use case B>
          <step #.> Replace |<modelElement A>
                   with
                                <modelElement B>
                         [ ]
                                <modelElement B>
```

Figure 3-43 [44]: Composition of Use Case A and Use Case B with Instantiation of |<modelElement> A Role.

3.4.3.3.3 Aspectual Use Case Driven Approach Process

The steps of the process are depicted in Figure 3-44. The process commences with actors and use case identification, as per the common Use Cases approach [8]. This is followed by refinement of the identified use cases to externalise the functionalities that are spread across several use cases; these functionalities are encapsulated in *include* or *extend* use cases.

Having modularised the functional concerns, the process addresses the identification of the non-functional ones by analysing the already elicited requirements as well as obtaining additional information from the system stakeholders. These NFRs are represented in a template (as shown in Figure 3-39).

[43] Source of Figure: [47] Figure 3.
[44] Source of Figure: [65].

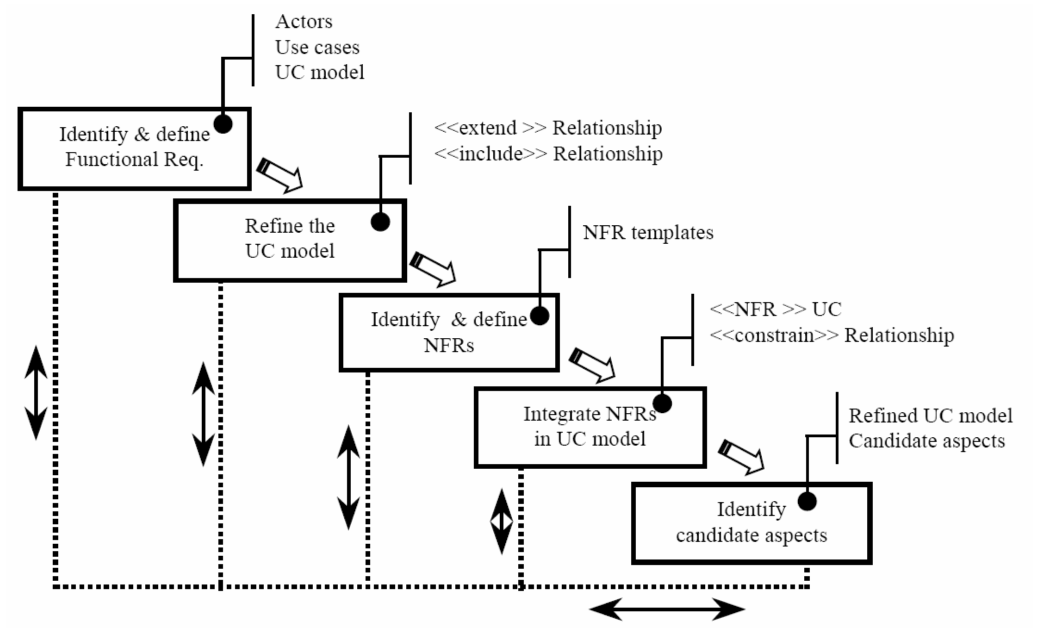


Figure 3-44 [45]: The process model for the Aspectual Use Case Driven Approach.

At the next step, the identified NFRs are integrated into the use case model derived in the previous steps. The integration is achieved by extending the use case model with the new stereotyped relationships which link the functional use cases with the new NFR use cases for each of the identified non-functional concern. The extended set of relationships, as discussed in section 3.4.3.3.2 (i.e. *extend, include, inherit, collaborate, damage, constrain*), is used.

Finally, the process summarises the *candidate aspect* use cases: if a use case is related to more than one other use case, then that use case is a good candidate for an aspect.

3.4.3.3.4 Identification and Treatment of Crosscutting Concerns with Aspectual Use Case Driven Approach

In this approach the functional requirements are identified through use cases. The use cases are then analysed for repeated behaviour and the crosscutting functionality is separated from all use cases into *included* or *extending* use cases. Also, a single-standing functionality can be identified as potentially crosscutting if it has relationships with many other use cases. Treatment of functional requirements is supported via the use cases approach, as well as can be assisted with the newly introduced additional relationships (damage, collaborate, constrain).

Identification of crosscutting NFRs, on the other hand, is not supported with a clear procedure. The NFRs are identified via analysis of other requirements and additional information. Once identified, the NFRs receive a systematic treatment: by being represented via templates which help to record their inter-relationships, via use case diagrams with quality attributes and generalised use case pattern and action specifications.

3.4.4 Multidimensional Separation of Concerns Approaches

Multidimensional Separation of Concerns [69] is based on the premise that all software development artefacts are made up of multiple, overlapping concerns and that software

[45] Source of Figure: [47] Figure 1.

development will benefit if software systems can be flexibly decomposed and composed according the alternative combinations and organisations of concerns [48].

Approaches discussed in this sub-section maintain the principle that concerns are worth studying by themselves and that no concern is more important than the others, i.e. there is no evident base and crosscutting distinction between concerns. Since all concerns are treated uniformly, one can project their influences on each other and hence observe and analyse any crosscutting behaviour amongst them.

3.4.4.1 Concern Modelling with Cosmos

Cosmos [48, 49, 70] is a general-purpose concern modelling schema which allows to model concerns generically and independently (irrespective of any other software artefact).

3.4.4.1.1 Concern Modelling with Cosmos Method

This work aims to promote concerns to be first-class entities in software development [50] where concerns are defined as any matter of interest in a software system [49].

The motivations for this intent are many. To illustrate some we should consider that concerns arise at all stages of software life cycle and are present in various forms throughout; a single concern can span multiple phases of the life cycle, be related to multiple instances and types of artefacts, and affect the phases and artefacts in different ways. Concerns also change over time, causing change in their respective software artefacts [49].

Cosmos models the software concerns space in terms of *concerns, relationships, predicates,* and *topics.*

Concerns are divided into two main categories: logical and physical. Logical concerns relate to conceptual entities, such as issues, problem, "itlities"[46], etc. Physical concerns refer to actual constituents of software systems, such as software units, hardware, services, etc. Each type of concerns can further be grouped into sub-categories [71].

Relationships reflect how the concerns interact with each other; they also are subdivided into several types. Predicates represent integrity conditions over various relationships and can be classified accordingly too [49]. Finally, topics are groupings of all other elements that relate to some particular topic of interest. The general concern model elements for Cosmos, as presented in [49] are depicted in Figure 3-45.

[46] i.e. such quality attributes as availability, reliability, etc.

- Concerns
  - Logical
    - Classifications
    - Classes
    - Instances
    - Properties
    - Types
  - Physical
    - Collections
    - Instances
    - Attributes
  - Predicates
  - Relationships
  - Groups
  - Predicates
    - // subtypes not elaborated

- Relationships
  - Categorical
  - Classification
  - Generalization
  - Instantiation
  - Characterization
  - Topicality
  - Attribution
  - Membership
- Interpretive
  - *Contribution*
  - *Motivation*
  - *Admission*
  - *Logical implementation*
  - *Logical composition*
  - *Logical requisition*
- Physical
  - Physical association
  - *Physical requisition*
- Mapping
  - Mapping association
  - *Physical implementation*

Figure 3-45 [47]: Cosmos Concern Model Elements: Outline.

3.4.4.1.2 Concern Modelling with Cosmos Artefacts

Cosmos intends to model the concerns independently of any type of artefact, consequently it does not produce any specific artefacts, but only concern descriptions, as illustrated in Figure 3-46.

The example shown in Figure 3-46 is that of a sample of concerns for a general purpose cache [70] which supports usual caching and also has some extra functionality such as logging, collection of statistics, and object dependency tracking. It is interesting to note, that in this example the cache has been implemented before its concerns were modelled, consequently, concerns collected for the cache also include its implementation related elements.

Logical Concerns: Classes

- Object classes
  - Cache
  - CachedObject
  - *Other object classes …*
- Functionality
  - Cache
    - Core
    - Object expiration
    - Operation-enabling
    - Statistics logging …
  - CachedObject
    - Core
    - Expiration
    - *Other functionalities …*
  - *Other classes …*
- Behavior
  - Cache
    - Operational
      - Core
      - Object expiration
      - Operation logging
      - Statistics logging…
    - Aspectual
      - Input checking
      - Operation logging..
    - Non-operational
      - Statistics logging
- State
  - Cache
    - Objects
    - Dependencies
    - Configurable controls …
- Properties
  - Static properties
  - Dynamic properties
- Java code
  - Programmed classes
    - Classes
    - Members
- Decomposed classes
  - *Anticipated*

Logical Concerns: Other Categories

Instances
- Omitted for brevity
- Topics
- Dependencies and transitivity
- Configurable behaviors
- Other topics ...

Properties
- Generality
- Performance
- Information hiding
- Concurrency
- Configurability
- Correctness
- Other properties ...

Physical Concerns

Instances
- com.ibm.ws.abr.gps.-
- Cache.java
- com.ibm.ws.abr.gps.-
- Cache.class
- com.ibm.ws.abr.gps.-
- CachedObject.java
- com.ibm.ws.abr.gps.-
- CachedObject.class
- Other classes ...

Collections
- whimbrel.watson.ib m.com
- C:\$Sutton\Caching\ Code\Java\Prog
- rammed
- Other collections ...

Attributes
- com.ibm.ws.abr.gps.
- Cache.java.Size
- Other attributes ...

Figure 3-46 [48]: Selected Concerns from GPS Cache modelled in Cosmos.

Figure 3-46 illustrates what concerns are involved, their functionality, behaviour, properties, possible mapping to a specific development language, etc. The concern space thus modelled is multidimensional, in that many elements in the cache can be

[47] Source of Figure: [49]] Table 20.1.
[48] Source of Figure: [49] Table 20.2 .

assigned to concerns in multiple classifications, and any of the available classifications can be selected as the *base* for viewing the rest of the concerns [49]. Once modelled, concerns can then be used in further software development artefacts at any stages of development.

By relating concerns to the various artefacts in which they appear, the developers can improve traceability, foresee impact of concern change and achieve many other benefits of modularisation and clear understanding of software artefact dependencies.

3.4.4.1.3 Concern Modelling with Cosmos Process

The concern model can be built for any type of development: new, evolutionary, COTS [49]. For instance, the example described above showed that concerns can be modelled for an already implemented system. In this case the development will probably be linked to the needs of evolution, or product family development. The concerns identified from the implemented system will be complemented with the concerns representing additional or changed features, which will be related to additional artefacts and their interactions with the existing system. The developed system can also be re-modularised on the basis of concern-units.

When used with COTS development, concern models can be used to evaluate the suitability of particular products for development from the perspective of how their concerns will interact with the concerns of other products, and contribute to the desired software system concerns, etc.

If concern modelling is used for a new development, a limited concern model can be built at the start of the development which can be elaborated as the development goes on. Concerns related to requirements, design etc. can be added and related to the artefacts, the relationships between these concerns can be analysed and the software products structured to reflect the semantics of concern interaction. This, in turn, facilitates evolution and change. The changes themselves too can be validated against the concern model.

Besides the descriptions above, presently there is no standard process defined for concern modelling and analysis with Cosmos [48]. The basic approach is to carefully study the available documents and artefacts of interest and identify the explicitly expressed concerns, as well as concerns implied by the documents and artefacts. It is necessary to select only concerns that are reasonably relevant to the software system, as the number of concerns will be very high even for small systems. The identified concerns then should be fitted into the Cosmos categories and used for whatever development type is appropriate.

3.4.4.1.4 Identification and Treatment of Crosscutting Concerns with Concern Modelling with Cosmos

Cosmos does not distinguish either between crosscutting and non-crosscutting or functional and non-functional concerns. All concerns are identified and modelled in the same way; they simply are classified per different categories of the schema.

3.4.4.2 Concern-Oriented Requirements Engineering

The Concern-Oriented Requirements Engineering Model (CORE) [13, 51] is an adaptation of the Aspect-Oriented Requirements Engineering (AORE) approach [1] discussed in section 3.4.1.1. While many of the requirement treatments and representation techniques used in AORE [1] are carried forward to CORE, the main difference between these approaches is the adoption of a uniform view on all concerns in CORE. *Concerns* in CORE imply any coherent collection of requirements.

3.4.4.2.1 Concern-Oriented Requirements Engineering Method

CORE strives to decompose requirements in a uniform fashion regardless of their functional or non-functional nature. This makes it possible to project any particular set of requirements on a range of other requirements, hence allowing the flexibility to choose "base" and "crosscutting" concerns as desired, rather then having to follow "functional base and non-functional crosscutting" tradition.

Figure 3-47 illustrates this model. It represents the CORE concern space at the requirements level: it is presented as a hypercube each face of which stands for a particular concern. Since all concerns are treated equally, any set of concerns can be selected as the base to project the influence of another concern or set of concerns onto this base. In this way CORE supports multi-dimensional separation of concerns [69].

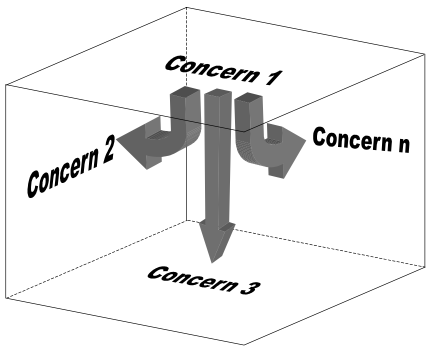


Figure 3-47 [49]: Concern space represented as a hypercube in CORE. The block arrows represent concern projections.

The concern projections are achieved through the same composition approach as in [1]: employing informal, often concern specific, actions and operators. The CORE approach also supports establishment of early trade-offs among crosscutting and overlapping requirements, as well as negotiation and decision-making among stakeholders using similar techniques to AORE [1]. Additionally, this model defines the notions of *meta concern* space and *compositional intersection* [13].

The *meta concern* space (Figure 3-48) comprises all abstract concerns (both functional and non-functional) that can manifest themselves in various systems. In this sense the meta concern space is a catalogue of concerns that appear time and again in various software systems. Consequently, the abstract concerns from this space can be used as a basis to classify the requirements of a given system in order to identify the corresponding concrete concerns in the system space. The utility of the meta concern space is in cataloguing the abstract concerns, their likely relationships and possible ways of utilisation.

[49] Source of Figure: [51] Figure 1.

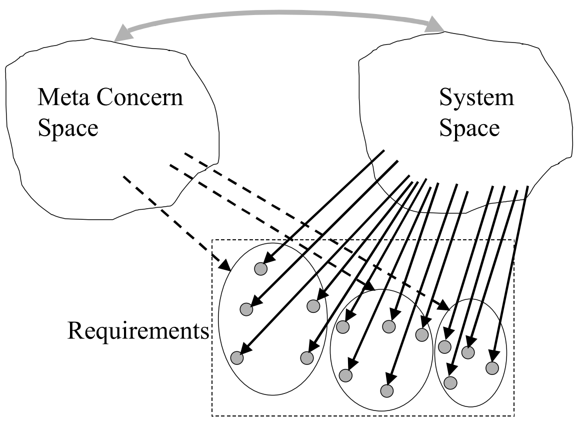


Figure 3-48 [50]: Meta Concern Space

The analysis of concern interactions and trade-offs is fairly straight forward in approaches which have a strong base-aspect separation. The base concerns act as a reference point with respect to which the trade-offs and interactions of aspectual concerns are analysed (e.g., [1], [44], etc.). However, in a multi-dimensional RE model, such as the one employed by CORE, each concern can affect multiple other concerns and one can choose any set of concerns as a base to observe the trade-offs amongst other concerns. If left unchecked, this can lead to serious scalability issues due to the potentially large number of concern combinations to be analysed. The notion of a *compositional intersection* introduced by [13] is, therefore, a very powerful concept within the model. A compositional intersection provides a restricted set of concerns that can be used as a base for concern projection during concern interaction and trade-off analysis. In order to understand compositional intersection let us consider that C_1, C_2, C_3, …, C_n are the concrete concerns in the system requirements and Sc_1, Sc_2, Sc_3, …, Sc_n are the sets of concerns that each of them cuts across respectively. If one wants to identify the trade-offs (if any) between C_1 and C_2 then, in order to do this, one should take the compositional intersection of Sc_1 and Sc_2. However, note that a compositional intersection is not a simple intersection as in set theory. If C_a is a member of both Sc_1 and Sc_2, C_a will appear in the compositional intersection iff both C_1 and C_2 influence/constrain the same or overlapping set of requirements in C_a. That is, if C_1 and C_2 influence disjoint sets then C_a will not be in the compositional intersection. The compositional intersection is thus used to simplify concern interaction analysis by reducing the number of potential combinations of concerns to be used as a base.

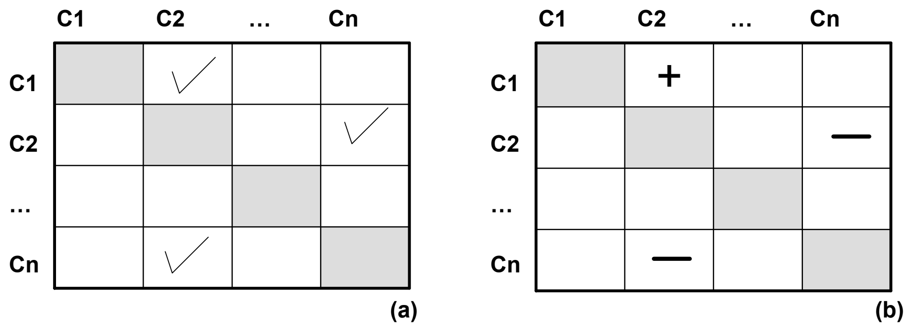
3.4.4.2.2 Concern-Oriented Requirements Engineering Artefacts

The concern representation in CORE is very similar to that of AORE. The concern artefacts are represented in XML (see Figure 3-25 for AORE) the only difference being that all concerns are encapsulated in <Concern> tag, rather than sorted into individual types, such as <Viewpoint> or <Aspect> etc. The hierarchical structure of a concern is same as for AROE: it can contain requirements and sub-requirements each of which has a unique identifier.

Similarly, the composition rules, operators, actions, and composition specifications illustrated in Figure 3-26 for AORE are also used for CORE, with only the <viewpoint> and <aspect> tags replaced by <concern>.

[50] Source of Figure: [13] Figure 2.

CORE, as AORE, also uses matrices for illustrating if the concerns influence each other (Figure 3-49 (a)) and also for depicting the types of contributions that concerns have on each other, where negative contributions indicate conflicts that need to be resolved (Figure 3-49(b)).



(a)

	C1	C2	...	Cn
C1	▨	✓		
C2		▨		✓
...			▨	
Cn		✓		▨

(b)

	C1	C2	...	Cn
C1	▨	+		
C2		▨		—
...			▨	
Cn		—		▨

Figure 3-49 [51]: (a) Table of concern relations; (b) Table of concern contributions

One artefact unique to CORE is the *folded table of contributions* which contains the reflected contributions of concerns towards each other. This is illustrated in Figure 3-50.

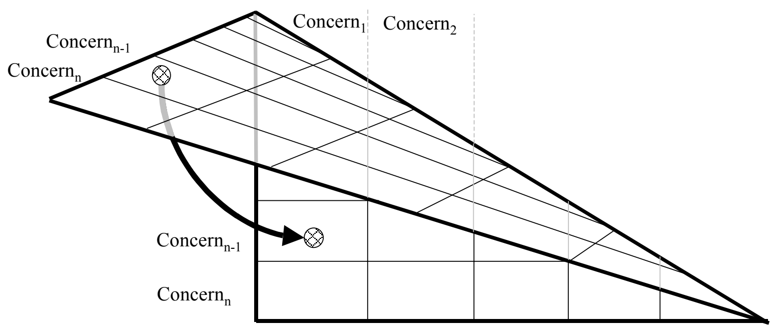


Figure 3-50 [52]: The concern contribution table folder along its diagonal.

Each cell in the folded table contains the reverse projections – note that the composition rules represent projections of the influence of one concern on other concerns – indicating how multiple concerns cumulatively influence a single one. This provides a powerful mechanism to observe both *influencing* and *influenced* relationships among concerns.

The conflicting concerns are allocated weights with respect to the concern for which the composition rule was defined and the concern with the higher weight gets priority during conflict resolution. In case if equal weights are assigned to conflicting concerns, stakeholder negotiations are necessary for conflict resolution.

3.4.4.2.3 Concern-Oriented Requirements Engineering Process

The CORE process is illustrated in Figure 3-51.

[51] Source of Figure: [51] (a) : Table 1; (b) Table 2
[52] Source of Figure: [51] Figure 3.

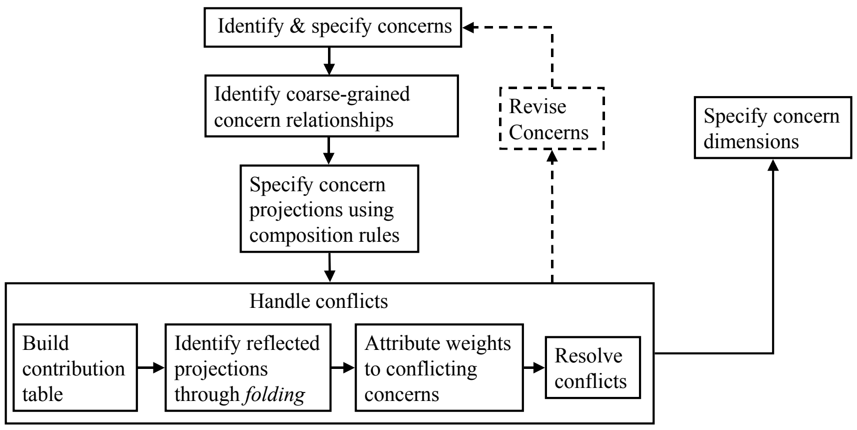

Identify & specify concerns
Identify coarse-grained concern relationships
Revise Concerns
Specify concern dimensions
Specify concern projections using composition rules
Handle conflicts
Build contribution table
Identify reflected projections through *folding*
Attribute weights to conflicting concerns
Resolve conflicts

Figure 3-51 [53]: The process model for CORE.

The process commences with concern identification which can be carried out with any mix of RE approaches, such as viewpoints or use case based approaches. Alternatively, or complementarily, the meta concern space can also be employed for concern identification. The identified concerns are related to each other through a matrix like one depicted in Figure 3-49(a). The relationships between concerns are identified using domain analysis, ethnographic studies, natural language processing or alike.

Having established the course-grained relations between concerns (Figure 3-49(a)), the more specific kind of influence between them is defined through composition rules. The composition rules apply at the requirements-level granularity of the concerns and are similar to those of the AORE approach, discussed in section 3.4.1.1.2.

After specifying the compositions between the concerns and their requirements, conflict identification and resolution begins with building of a contribution matrix (Figure 3-49(b)). This table is then folded to obtain the cumulative influence of concerns on each other (note that [51] and [13] propose two different approaches to folding as the latter employs a compositional intersection as the basis for trade-off analysis). Where a conflict is detected, a priority is assigned to the conflicting concern with respect to the concern that it is projected on. The decisions regarding conflict resolution can be then discussed with the stakeholders, using the assigned weights to assist with decision making. The conflict resolution might lead to requirement re-definition and iteration of the above process.

Once the conflicts are satisfactorily resolved, the mapping of the concerns onto the later stages of development is identified in the same way as for AORE. Additionally, the influence of concerns on the development lifecycle is specified: for instance, an availability concern may influence system architecture, while mobility may influence all: architecture, design, and implementation. This mapping and influence is considered in more detail in [13] which discusses the various architectural choices posed to the developer by each concern and its associated trade-offs. The architectural choices made by the developer for each concern then pull the architecture in various, at times conflicting directions. The trade-off and interaction analysis conducted at the requirements level helps to tailor these choices to ensure that the architecture has an optimal pull in the various directions and meets the stakeholders' requirements effectively.

[53] Source of Figure: [51] Figure 2.

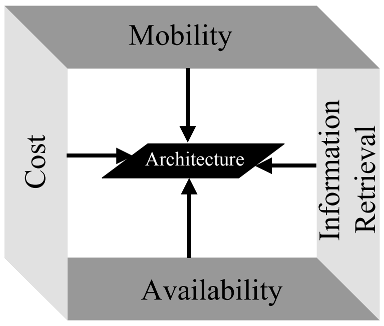


Figure 3-52[54]: Architectural pull of various concerns

3.4.4.2.4 Identification and Treatment of Crosscutting Concerns with Concern-Oriented Requirements Engineering

Similar to AORE, the CORE approach has a generic mechanism for concern handling suitable for both functional and non-functional concerns. Moreover, CORE does not make any particular distinction between these types. The approach also suggests a number of methods that can be used for concern identification as well as concern relationship identification. The meta concern space based concern identification approach can be complemented by the semantic natural language processing-base concern identification tool (discussed in section 3.4.2.1.4) which is presently under construction [59]. Such an integrated technique can provide a powerful concern identification mechanism well suited for both AORE approaches based on an aspect-base separation and those employing a multi-dimensional perspective for requirements analysis.

3.5 Component-Based AO

The AORE approaches discussed so far have not specified the granularity of the aspectual modules. The Component-Based Aspect-Oriented techniques address the utility of aspects within a component-based system, thus, applying the notion of aspects specifically to modules of a large granularity.

3.5.1.1 Aspect-Oriented Requirements Engineering for Component-Based Software Systems (AOREC)

Aspect-Oriented Requirements Engineering[55] for Component-Based Software Systems (AOREC) [52, 53] is devised to address such open issues of traditional component requirements engineering as classification of component services per systemic areas (or aspects) of application, sufficient detail per service, and ability to address a given service at required detail level at run-time. An *aspect* in AOREC is a characteristic of a system for which components provide or require services [52]. Aspects help to identify, categorise, and reason about the component requirements.

3.5.1.1.1 AOREC Method

AOREC focuses on identifying and specifying the component requirements relating to key *aspects* of the system. Examples of such systemic aspects are, for instance, user

[54] Source of Figure: [13] Figure 11.
[55] The term "aspect-oriented requirements engineering" has been for the first time mentioned in this work in [52].

interface, persistence, collaborative work, etc. Then they can be refined into sets of *aspect details*. For instance, a user interface can have the *affordance* and *view* details. Such *aspects* are a way to take multiple, systemic perspectives onto components and, in so doing, to better understand and reason about the data, functionality, constraints, and inter-relationships of components. For instance, they can be helpful in reasoning about how different components interact via providing and requiring *aspect details* (e.g., see Figure 3-53), or what will be required when reusing a specific component, or if a given configuration of components will be valid with respect to a given *aspect* of the system, etc.

3.5.1.1.2 AOREC Artefacts

The aspects can be produced for both newly developed and re-engineered components. In Figure 3-53 an example of aspect-oriented requirements identification artefacts obtained through re-engineering is provided. The figure demonstrates that an Event History component provides and uses with several aspects, namely: User Interface, Persistence, Collaborative Work and End User Configuration aspects. Each aspect details within an aspect can be provided or required by a given component. For instance, in Figure 3-53 extensible affordance detail in the User Interface aspect is provided by the Event History component, but this component also requires viewer detail from the same aspect. The viewer detail is provided to the User Interface aspect by another component: the Event History Viewer.

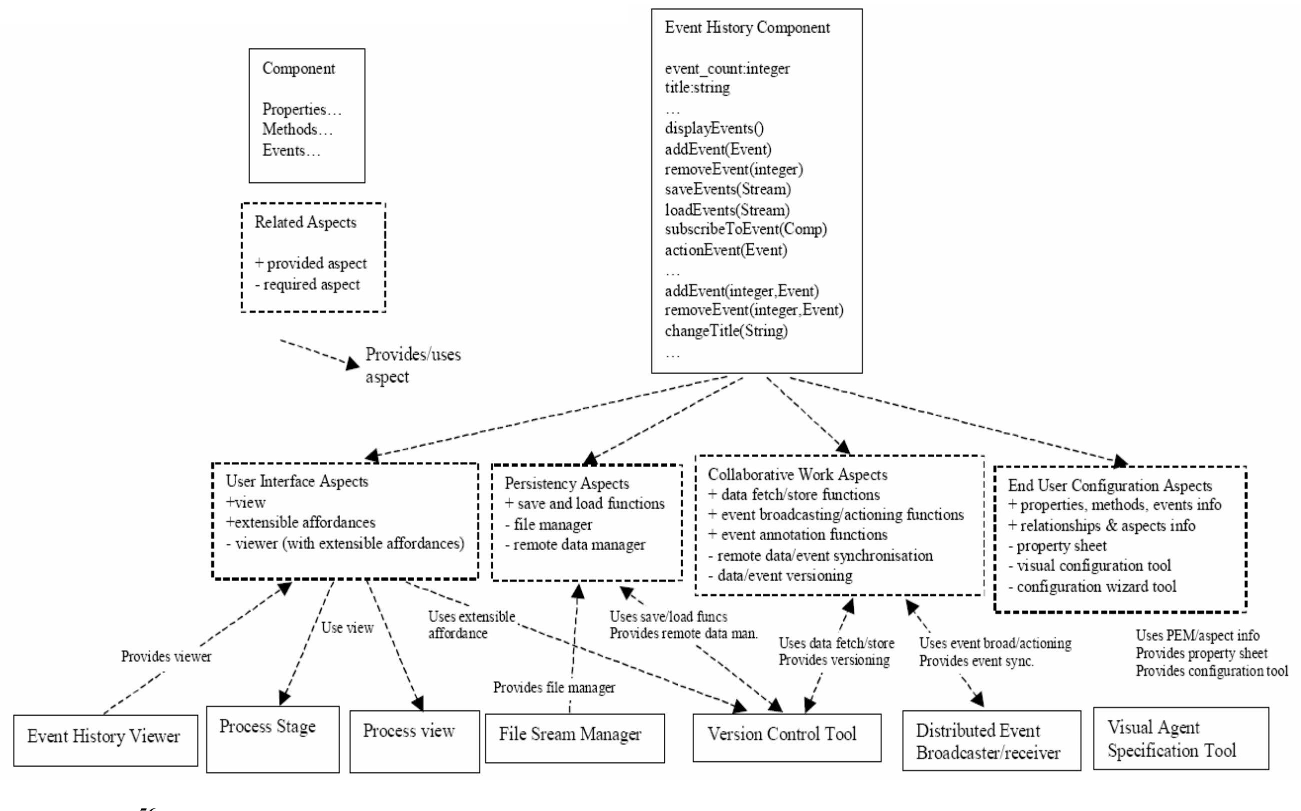


Figure 3-53 [56]: Example components and some of their aspects.

Aspects identified and documented with diagrams, such as in Figure 3-53, are complemented with detailed textual descriptions. These descriptions provide additional documentation for functional and non-functional requirements. The full textual aspect-oriented specification for the Event History component is provided in Figure 3-54.

[56] Source of Figure: [52] Figure 4.

Collaborative Work Aspects : COLLABORATION

1) +data fetch/store functions : DATA_MANIPULATION

-- Provides services for getting some/all of event history data and for updating some/all of event history data. Used by components providing collaborative work infrastructure to keep

distributed data synchronised or partially synchronised.

QUERY=true; UPDATE=true

2) +event broadcasting/actions functions : EVENT_MANAGEMENT

-- Provides services allowing other components to detect event history update events and to action (replay) events received by other components. Used by components providing collaborative

work infrastructure to keep distributed event history synchronised or support deltas of event history version changes.

DETECT=true; ACTION=true

3) + event annotation functions : AWARENESS

-- Provides services for annotating, selecting, highlighting events. Used by components providing collaborative work infrastructure to support basic group awareness facilities for updated

event history events. Other components should use these to annotate events with remote user name, colour them with a colour associated with a particular user, etc.

HIGHLIGHT=colour; ANNOTATE=tex

4) - remote data/event synchronisation : LOCKING

-- Requires component(s) that supports remote data/event synchronisation. Could support fully synchronised data or semi-synchronous update. This should be robust if network connections

fail, and should work over low or high bandwidth networks.

SYNCHRONOUS=true OR false; SEMI_SYNCHRONOUS=true OR false; NETWORK_SPEED=any; STORE=true

5) - data/event versioning : VERSIONING

-- Requires component(s) providing data versioning. Should support both event history data and event history update event recording/versioning. This should be a simple-to-use facility for

end users. Should extend the viewer affordances to provide at least check-in/check-out capabilities via +extensible affordance aspect.

DATA=true; EVENT=true; INTERFACE=extensible affordances; CHECKIN=true; CHECKOUT=true

Figure 3-54 [57]: Detailed aspect-oriented component requirement specifications.

AOREC also introduces the concept of *aggregate aspect* which is an aspect specified for groups of interrelated components used to reason about aspect-oriented requirements of a set of components or even the whole system.

3.5.1.1.3 AOREC Process

The AOCRE process is demonstrated in
Figure 3-55. The process starts with analysing general application requirements. The system requirements are used to identify candidate components, as shown by step (1) in

[57] Source of Figure: [52] Figure 5.

Figure 3-55. The requirements for identified components are then elaborated. Aspects for each component are identified (step 2) and refined to determine the provided and required aspect details (step 3). Aspects are used to reason about component composition and configuration.

The refined aspects for groups of components, or the whole system, if appropriate, are analysed for *aggregate aspects* (step 4). The aggregate aspects can then be identified as new components, thus also causing change in previously analysed components and initiating revision cycles (step 3.b).

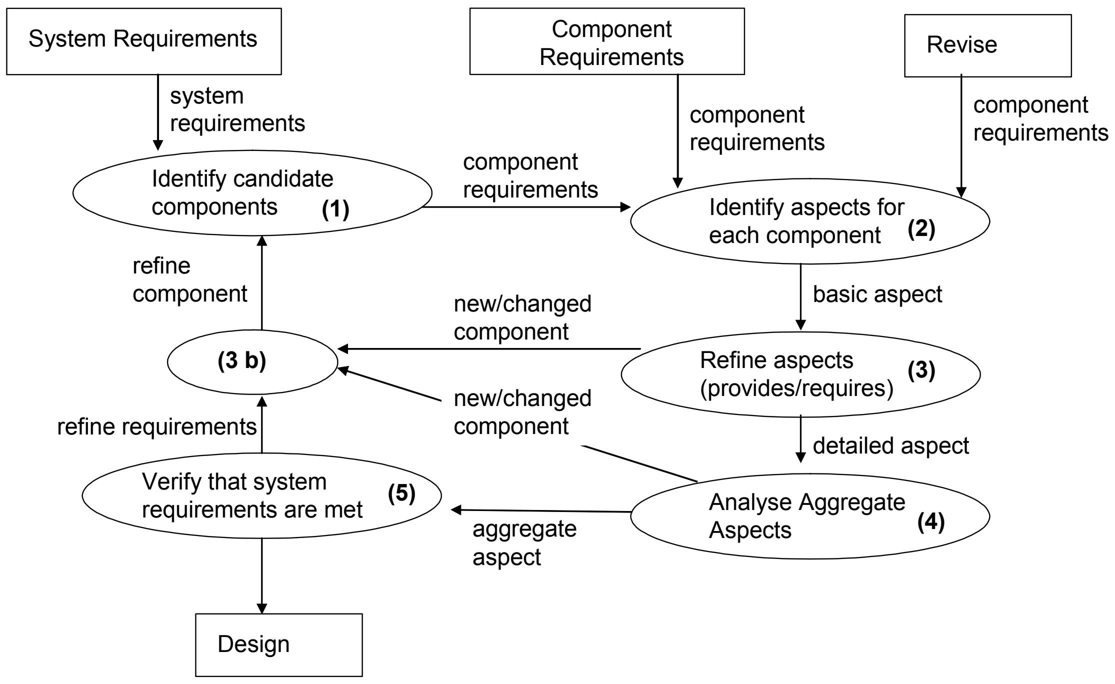


Figure 3-55[58]: Basic AOREC process.

Once all system requirements are allocated per components; aspects for components and component groups are identified and aggregated, the produced components and aspects are verified against the system requirements (step 5). If the requirements are satisfactorily met by the produced component and aspect requirement model, the design phase commences.

3.5.1.1.4 Identification and Treatment of Crosscutting Concerns with AOREC

AOREC does not provide any general support for crosscutting concern identification. The aspects and aspect details are identified on a case-by case basis by the requirements engineer.

For a small subset of aspects (namely User Interface, Collaboration, Persistence, Distribution, and Configuration) used in case studies for publications on AOREC [52, 53] an initial reusable breakdown of aspects to aspect details is suggested.

[58] Source of Figure: [52] Figure 3.

This approach is well suited for treating component contributions to non-functional concerns, as many (all) components will contribute to any given non-functional requirement. Examples of such non-functional aspects are provided in the AOREC work. On the other hand, it is difficult to perceive how a functional requirement could be crosscutting in a component-based system, where components are intended to modularise functionality. Presently there are no examples of functional concerns for AOREC. Yet, the general approach of using provided/required aspect details for an aspect is well suited for both non-functional and functional aspects.

3.5.2 Other AO Approaches

We have discussed that most AORE approaches have emerged as extensions of some non-AO approaches. Yet, there are also a small number of AORE approaches which are completely new or are driven by the needs of existing AO approaches at design and implementation level. Currently, the most prominent such approach is Theme/Doc. Although the Theme approach has emerged from the work on Subject-Oriented Programming [72], Theme/Doc itself has not: it is a new approach developed to provide requirements analysis capabilities for subsequent aspect-oriented design with Theme/UML.

3.5.2.1 Theme/Doc

Theme/Doc [54-56] is the Requirements Engineering part of the Theme [55, 56] approach. The core of the approach is the concept of a *theme* which represents a meaningful unit of cohesive functionality.

The *themes* are loosely similar to functionalities identified by use cases. The method is centred around graph-based representation of the potential themes and this representation-assisted analysis.

3.5.2.1.1 Theme/Doc Method

Them/Doc [54-56] supports "aspect identification and analysis in requirements documentation "where aspects manifest themselves as "descriptions of behaviours that are intertwined, and woven throughout" [55]. Thus, it is aimed at later stages of RE, when at least an initial requirements document is available for lexical analysis. Due to its focus on "behavioural" aspects, it aims to discover the functional crosscutting concerns, since the non-functional ones could have no behaviour-related manifestations.

The approach is supported by the Theme/Doc tool which provides a set of views that assist in requirements analysis, as well as direct mapping of the requirement views to Theme/UML – the design counterpart of the Theme approach. In fact, the Theme/Doc tool is quite central to the Theme/Doc approach because the analysis and steps of the approach are based on the graphical visualisations from the tool.

Currently the work on Theme/Doc is focused on addressing the scalability issue of the approach [73]. It is also looking at other possible clues for detecting crosscutting, in addition to the current one of having several actions/concerns mentioned in the same requirement.

3.5.2.1.2 Theme/Doc Artefacts

Most artefacts produced by the Theme/Doc approach are those generated by the Theme/Doc tool. However, the initial input to the tool is the manually compiled list of *action words* and *entities*. *Action words* are the verbs from the requirements document which indicate some activities performed, while *entities* are the nouns to which those activities relate directly or indirectly.

The central generated artefact of the approach is the *action view* graph depicted in Figure 3-56 for a course registration system where students are registered/unregistered for a course and marks given to them:

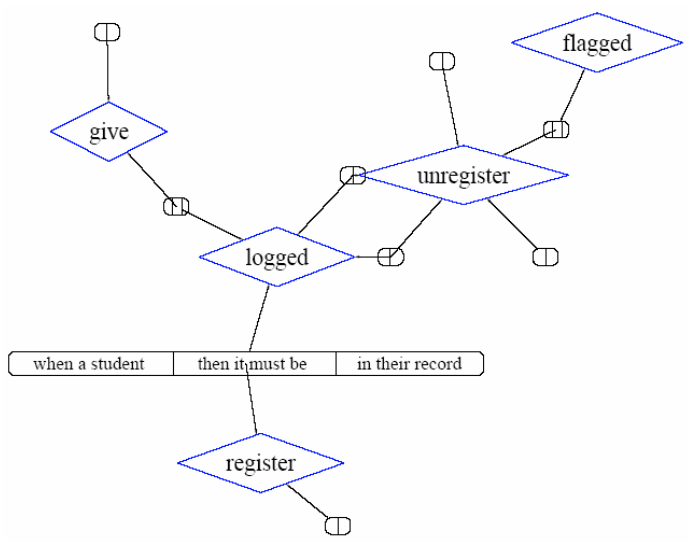


Figure 3-56 [59]: Action View in Theme/Doc for a Course Registration System.

When the action words are assigned per requirement and the links between secondary actions and their requirements are cut (or *clipped*) and replaced with a decorated link, the *clipped action view* is produced (Figure 3-57 (a)). Besides identifying the *base* and *crosscutting* themes (*base* ones being the themes at the lowest level of the clipped action view hierarchy, and *crosscutting* ones being those at the higher levels), this view also demonstrates the required order of composition. The themes should be composed from the bottom up, as the higher level themes might rely on those below them.

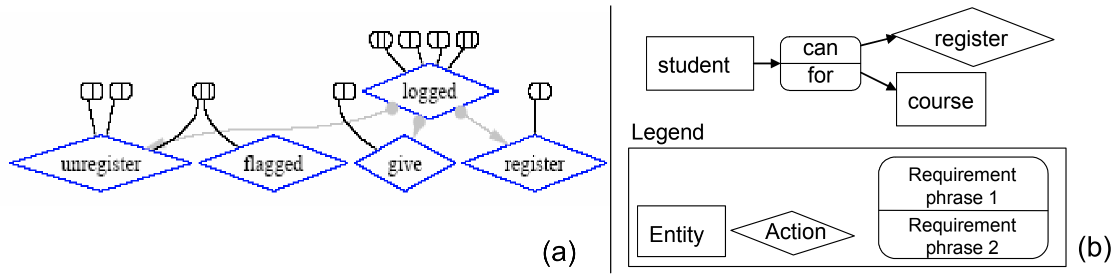


Figure 3-57 [60]: Clipped Action View (a) and Theme View (b) From Theme/Doc Tool for a Course Registration System.

The actions and their associated entities and requirements are collected into a theme which can be viewed in the theme view (Figure 3-57 (b)).

[59] Source of Figure: [55] Figure 1.
[60] Source of Figure: [55] (a) Figure 2, (b) Figure 3.

3.5.2.1.3 Theme/Doc Process

The process of Theme/Doc [55, 56] is shown in Figure 3-58.

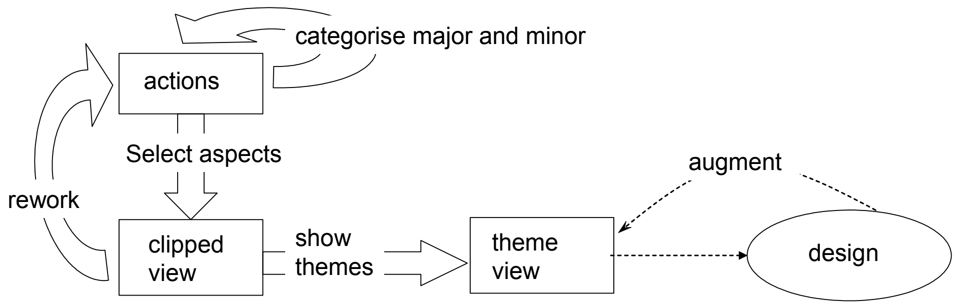


Figure 3-58[61]: Theme/Doc Process

It commences with the requirements engineer identifying *action words* from the requirements document and providing these, along with the requirements document itself, as inputs to the Theme/Doc tool. It should be noted that synonyms referring to the same action can be grouped under one action word, and so can "minor" actions which do not have strong enough themes of their own.

Using the provided inputs the tool generates *action views*. This view demonstrates the links between requirement sentences and the action words, as shown in Figure 3-58. If a requirement is linked to many action words, tangling of behaviour is identified. This could be caused either by a badly specified requirements (in which case the specification should be corrected), or by presence of crosscutting behaviour.

In case of many links from a requirement to action words, where re-stating requirements does not alleviate tangling, the predominant action for the requirement should be defined (and thus selected as *base*), and the secondary action(s) should be *clipped* (using the tool), which indicates that the secondary action/behaviour will crosscut the base behaviour.

After all shared requirements are clipped from their secondary behaviours (with a faded arrow between primary and secondary behaviours replacing the link between the requirement and the secondary behaviour), the *clipped action view* is obtained (e.g., Figure 3-57 (a)).

The *theme* view is created by identifying entities, to be used at design stage, from the requirements document and providing these as additional input along with the actions words and the requirements document. The theme view can be used for the design stage to produce separate themes for mapping onto Theme/UML. This view also helps to design generic views of the crosscutting themes (by generalising the concrete project-related specifics from the crosscutting theme).

At this stage the work of Theme/Doc part of the Theme approach is completed and Theme/UML steps in to create the designs. However, after the design process is complete, the final designs can be validated against the theme views of Theme/Doc. The theme views can also be augmented by design level elements. This can help to verify that the produced designs align with the requirements.

[61] Source of figure: adopted from [56]

3.5.2.1.4 Identification and Treatment of Crosscutting Concerns with Theme/Doc

As discussed above, Theme/Doc approach has based aspect identification on using action words, which makes it well suited to identify crosscutting functional requirements. On the other hand, non-functional requirements often do not have any action associated with them. The approach suggests that in such cases the requirements can be re-written to include action words. However this assumes that such requirements can be identified by the requirements engineer which in most cases is precisely the problem that the identification needs to address. Besides, the issue of how a particular non-functional requirement is related to other requirements is still unresolved (e.g., how security affects response time of mark allocation).

3.6 Comparison

Having outlined the RE approaches earlier in this section, we now investigate how well they perform against our comparison criteria (presented in section 2 and refined in section 3.2).

3.6.1 Traceability through software lifecycle

3.6.1.1 Traceability of Requirements and Change to Their Sources of Origin

Table 3-1 shows the comparison of the features that support traceability of requirements and change to their sources of origin for all the non-AO and AO approaches considered in section 0.

Approach	**Features That Support Traceability Of Requirements And Change To Their Sources Of Origin Criterion**
PREview	source and change history sections in template; use of viewpoint focus
VIM	viewpoint owner, work record section in a viewpoint
NFRF	SIG for sub-goals from the softgoals, indirectly: softgoal topic
Problem Frames	indirectly: context and problem diagrams and problem frames.
KAOS	domain model, links between agents and goals; gaol decomposition graphs, formal representations
I*	SD and SR diagrams, within SR diagrams the SIGs for goal and softgoals from the softgoals; implied intentionality
Use Cases	actor in the use case diagram
Misuse Cases	actor in the use case diagram, including misuse cases and solutions to mitigate them.
AORE with Arcade	viewpoint of requirements
ARGM	SIG for sub-goals from the softgoals, indirectly: softgoal topic
AOSD/UC	actors in use case diagram
SMA	actors in use case diagram, source in templates for NFR
AUCDA	actors in use case diagram, source in templates for NFR

Cosmos	not considered
CORE	abstract concerns in the meta concern space
AOREC	not considered
Theme/Doc	not considered

Legend: PREview (section 3.3.1.1); VIM: Viewpoints and Inconsistency Management (section 3.3.1.2); NFRF: Non-Functional Requirements Framework (section 3.3.2.1); KAOS: (section 3.3.2.2); I* (section 3.3.2.3); PF: Problem Frames (section 3.3.3); Use Cases (section 3.3.4.1); Misuse Cases (section 3.3.4.2); AORE with Arcade: Aspect Oriented Requirements Engineering with Arcade (section 3.4.1.1); ARGM: Aspects in Requirements Goal Models (section 3.4.2.1); AOSD/UC: AOSD with Use Cases (section 3.4.3.1); SMA: Scenario Modelling with Aspects (section 3.4.3.2.1); AUCDA: Aspectual Use Case Driven Approach (section 3.4.3.3); Cosmos (section 3.4.4.1); CORE: Concern Oriented Requirements Engineering (section 3.4.4.2); AOREC: Aspect Oriented Requirements Engineering for Components (section 3.5.1.1); Theme/Doc (section 3.5.2.1).

Table 3-1: Summary of Features that Support Traceability of Requirements and Change to their Sources of Origin Criterion.

PREview records the source of a requirement origin in the *source* section of requirement templates and keeps the record of change in the *change history* section. Besides, through the reference of a requirement to the viewpoint that it comes from, one can use the viewpoint *focus* for relating the requirement to the part of the system that its originating viewpoint represents.

In VIM each viewpoint and its requirements are assigned to a *viewpoint owner* for whom the viewpoint is elicited. Also the *work record* section of each viewpoint will contain the development and change state and history.

Although NFRF records the origin of sub-goals from the softgoals in the SIG, it does not record the sources of softgoals themselves. Using the topic of the softgoal, one can get a general idea as to where in the system it belongs, but not where the softgoal originated from.

PF does not explicitly attend to this issue, yet in many cases the sources of the requirements may be identified from the context and problem diagrams and problem frames.

The conceptual model in KAOS provides a way of detailing the relationships between goals the agents from which they originate through the kinds of links characteristic for a given domain. On the other hand, the origins of sub-gals are traceable back to the larger goals through the goal decomposition graphs. Additionally, all information in KAOS is formally represented and recorded, thus providing another traceability link.

I* identifies the agent associated with each goal, task or resource. Thus, each elicited requirement will have an associated agent. Even the reason for the requirement or change is identifiable from the SR diagrams. Also, since I* supports "implied" intentionality, non-human agents (such as government bodies, law, etc.) will act as agents when they generate requirements or cause change.

Use Cases identify the functionality per an actor, and record the actor and his/her use cases (and their related requirements) in the use case diagram. This approach does not record the information related to non-user level requirements and change.

Misuse Cases, also record the actors and use cases, in the same way as the Use Case approach does. In addition, actors here can be people, external system, internal sub-systems, and inanimate objects and phenomena. In this way the non-user level requirements and change can also be recorded. This, however, necessitates design and even implementation level requirement integration with the user-level requirements.

AORE with Arcade keeps clear reference to the sources of requirements origin, as each requirement is nested within the viewpoint that it originates from. It is unclear if and how AORE with Arcade will record the sources of change, as no example or discussion on this is available. However, we envisage that version control in the native XML database system employed by the Arcade tool can be used for the purpose.

Aspects in Requirements Goal Models approach uses the NFR-type recoding, thus it has the same shortcomings as the NFR framework in recording sources of origin for goals, softgoals and change.

Similar to Use Cases, AOSD/UC records the origin of functional requirements on the use case diagram, though when classifier representation is used instead of ellipse, these details might be lost. Additionally, this approach records the non-functional concerns through a parameterised <Perform Transaction> use case for a parameterised <Actor>. The parameterised use case is instantiated for each relevant actor type, thus recording the source of origin of specific sets of non-functional concerns per actor.

Although the Scenario Modelling with Aspects approach does not detail how source traceability is preserved, it relies on use cases for requirement identification and scenario generation, which provides the mechanisms of the Use Cases approach. In addition, the templates used for non-functional requirements [46] record their source.

In the Aspectual Use Case Driven approach the functional requirements are traceable to their sources through the use case diagram and non-functional ones through the templates used for their representation.

In Concern Modelling with Cosmos the source of the concerns is not deemed important, as many concerns will be implied.

Unlike AORE, CORE does not use viewpoints for concerns structuring. However, the sources of concern origin are recorded via the classification based on the abstract concern representations in the meta concern space.

AOREC does not address the issues of requirement or change origin.

Theme/Doc assumes that some requirements document is already produced and does not attend to the issue of the requirements source or the source of change to the requirements. Such issues are assumed to be handled during the initial requirements document production.

3.6.1.2 Traceability between Lifecycle Artefacts

Table 3-2 shows the comparison of the features that support traceability of requirements between lifecycle artefact representations for all the non-AO and AO approaches considered in section 0.

Approach	**Features That Support Traceability Between Lifecycle Artefacts Criterion**
PREview	unique identifiers link requirements to viewpoints
VIM	viewpoint work plan , templates and specifications
NFRF	*design decision* and *operationalisation* and correlation links, claims and augmentations in SIG
KAOS	operationalisations, formal representation
I*	*design decision* and *operationalisation* and correlation links, claims and

	augmentations in goal and softgoal interdependency graph
Problem Frames	problem diagrams and annotations, *frame concerns*.
Use Cases	collaboration diagrams
Misuse Cases	records of misuse case solutions, collaboration diagrams,
AORE with Arcade	records of mapping decisions; PROBE framework
ARGM	*design decision* and *operationalisation* and correlation links, claims and augmentations in SIG
AOSD/UC	collaboration diagrams; use case slices and modules
SMA	not considered
AUCDA	collaboration diagrams
Cosmos	physical concern links to conceptual counterparts
CORE	records of mapping decisions and influence of concern on architectural choices
AOCRE	design-level aspects
Theme/Doc	direct match between its requirements engineering and design models; *major action* and *theme* views

Legend: PREview (section 3.3.1.1); VIM: Viewpoints and Inconsistency Management (section 3.3.1.2); NFRF: Non-Functional Requirements Framework (section 3.3.2.1); KAOS: (section 3.3.2.2); I* (section 3.3.2.3); PF: Problem Frames (section 3.3.3); Use Cases (section 3.3.4.1); Misuse Cases (section 3.3.4.2); AORE with Arcade: Aspect Oriented Requirements Engineering with Arcade (section 3.4.1.1); ARGM: Aspects in Requirements Goal Models (section 3.4.2.1); AOSD/UC: AOSD with Use Cases (section 3.4.3.1); SMA: Scenario Modelling with Aspects (section 3.4.3.2.1); AUCDA: Aspectual Use Case Driven Approach (section 3.4.3.3); Cosmos (section 3.4.4.1); CORE: Concern Oriented Requirements Engineering (section 3.4.4.2); AOREC: Aspect Oriented Requirements Engineering for Components (section 3.5.1.1); Theme/Doc (section 3.5.2.1).

Table 3-2: Summary of Features that Support Traceability between Lifecycle Artefacts Criterion.

PREview concerns end up scattered across viewpoints in the requirements elicitation stage and across requirements specification and design artefacts later on, though the requirements have unique identifiers that link them back to the concerns that they represent.

For artefact traceability VIM uses the *work plan* section of the viewpoint which provides such information as *viewpoint actions* and *trigger/guide actions* which respectively detail how and when new instances of a particular template should be created. Since the new instances may represent the later lifecycle stage specifications of a given artefact, a traceability link between them will be established (e.g., RE stage objects can become detailed objects at the design stage) [16, 74].

NFRF relates its softgoals to the appropriate functional requirements via *design decision* links while the *operationalisations* relate to design decisions via *operationalisation* links, thus, linking requirements and designs as well as functional requirements with the non-functional ones which they are related to. Besides, all design decisions and choices are recorded in the SIG via claim softgoals and augmentations which remain available during further development.

In KAOS operationalisations define the simple tasks to which goals are reduced. These tasks are assigned to corresponding agents and the pre and post conditions for respective actions of agents are defined. Although this does not directly trace to design entities, it may facilitate such traceability, for instance, by facilitating identification of classes per main agents, and grouping data and functionality for each agent in a class, etc.

While I* is mainly concerned with early requirements engineering process, some support for lifecycle artefact traceability is inherent within the method due to the use of NFRF-style goal and softgoal decomposition in its Strategic Rationale diagrams. From this perspective I* provides traceability support similar to NFRF. However, since I* encompasses goal decomposition (in addition to softgoal), functionality is more closely related to non-functional requirements then with NFRF.

Although PF does not provide explicit means for traceability preservation the approach is focused on understanding the links between the real world and the machine, which become documented in the problem diagrams and annotations. In addition, the solution patterns can be traced to the problem frames via *frame concerns*. However, the approach does not consider traceability needs in justification for selected problem decomposition, or problem frame selection, or variant extension.

The identified use cases in the Use Cases approach are mapped onto collaboration diagrams at the architecture design stage, but are consequently dissolved into object classes where all parts of a given class are combined from all use cases. This makes traceability of an individual use case to its specific contribution to a class difficult to maintain.

From the traceability between lifecycle artefacts perspective the misuse cases are the same as ordinary use cases. The only slight difference is that, in some simple cases, when a solution for misuse case mitigation is clearly defined on the use case diagram, it could be traced straight to the design or implementation. For instance, if the solution to the security requirement is identified as a training procedure for the system users, it will be clearly traceable to the requirement.

AORE with Arcade records what type of architecture or design artefact a concern transforms to (e.g., decision, function, etc.) and can trace it to architecture, design, and implementation via the PROBE framework [57]. The PROBE framework helps in tracing not only initial aspectual requirements, but also their associated trade-off.

For this criterion too the Aspects in Requirements Goal Models approach is same as the NFR framework, as it uses exactly the same mechanisms.

Similar to Use Cases, in AOSD/UC use cases are mapped onto collaboration diagrams at the architecture design stage, but here, thanks to use of use case slices, the partial information on operations and states of object classes can be preserved independently too.

Scenario Modelling with Aspects does not consider scenario tracing to the later stages, however the approach provides valuable insight into how aspects affect the future system behaviour.

Aspectual Use Case Driven approach does not address the traceability of its NFR use cases to the later stages of development, while the functional use cases can be traced to the collaborations, as in the standard Use Cases approach.

Concern Modelling with Cosmos is focused on concern modelling only. It does not venture into any development activity. Nevertheless, when the modelled concerns are represented as artefacts, these will be represented as physical concerns in Cosmos with links to their conceptual counterparts, in this way promoting artefact traceability.

The CORE approach records details of concern mapping to architectural decisions, design level function, or crosscutting design elements, as well as their scope of influence on the later development stages. Although presently the PROBE framework [57] is not yet adapted to be used for CORE, it may be adapted with not much change required. This will be addressed in the near future in order to help with tracing and validation of the initial aspectual requirements as well as their related trade-off decisions.

The requirements level aspects and aspect details identified by AOREC are directly propagated to design level, where these are refined.

Theme/Doc keeps clear links between requirements artefacts and their design incarnations due to direct match between its requirements engineering and design models as well as the *major action* and *theme* views of the Theme/Doc tool which allow to compare the requirements grouped into a theme against their designs.

3.6.2 Composability

Table 3-3 shows the comparison of the features that support composability of requirements all the non-AO and AO approaches considered in section 0.

Approach	**Features That Support Composability**
PREview	not considered
VIM	inter and intra viewpoint check rules
NFRF	indirectly: SIG
KAOS	indirectly: and/or groups in goal decomposition, formal representation, heuristics
I*	agents, dependency relationships, indirectly: SIG
Problem Frames	common domains
Use Cases	*extend* and *include* relationships and use cases
Misuse Cases	*extend* and *include* relationships with *mitigates, threatens, aggravates, conflicts with, has exception* stereotypes and use/misuse cases
AORE with Arcade	Flexible and extensible composition rules and operators, unique ids for viewpoints, requirements and sub-requirements
ARGM	matching topics of goals and task
AOSD/UC	*extension pointcut* ; *extend* and *include* relationships an use cases
SMA	binding definitions; operators for message interleaving; incoming and outgoing events; states; role parameters in IPS and SMPS specifications
AUCDA	new use case relationships: *collaborate, damage,* and *constrain* as well as standard *extend* and *include*; role parameters in UCPS and APS; binding specifications
Cosmos	concern relationships and constraint specifications
CORE	Flexible and extensible composition rules and operators, unique ids for viewpoints, requirements and sub-requirements, concern projections and compositional intersections
AOREC	per-component provided/required aspect details, provides/requires links
Theme/Doc	order for theme composition in *clipped action view*

Legend: PREview (section 3.3.1.1); VIM: Viewpoints and Inconsistency Management (section 3.3.1.2); NFRF: Non-Functional Requirements Framework (section 3.3.2.1); KAOS: (section 3.3.2.2); I* (section 3.3.2.3); PF: Problem Frames (section 3.3.3); Use Cases (section 3.3.4.1); Misuse Cases (section 3.3.4.2); AORE with Arcade: Aspect Oriented Requirements Engineering with Arcade (section 3.4.1.1); ARGM: Aspects in Requirements Goal Models (section 3.4.2.1); AOSD/UC: AOSD with Use Cases (section 3.4.3.1); SMA: Scenario Modelling with Aspects (section 3.4.3.2.1); AUCDA: Aspectual Use Case Driven Approach (section 3.4.3.3); Cosmos (section 3.4.4.1); CORE: Concern Oriented Requirements Engineering (section 3.4.4.2); AOREC: Aspect Oriented Requirements Engineering for Components (section 3.5.1.1); Theme/Doc (section 3.5.2.1).

Table 3-3: Summary of Features that Support Composability.

Problem decomposition is a natural way of reducing complexity, and it is indeed the path taken by all the approaches discussed above. On the other hand, the need for composing the decomposed requirements/concerns and understanding their mutual relationships has been addressed less thoroughly.

PREview decomposes stakeholder requirements into viewpoints and concerns, but does not consider the issue of composition. The details for each concern-viewpoint interaction are maintained for each viewpoint separately.

VIM does not directly address composition, instead it limits to the issue of consistency checking - a necessary, but not sufficient characteristic of composition. Moreover, while the approach demands that consistency checking is carried out, it does not provide any assistance with the difficult task of deciding which consistency rules need to be defined.

NFR Framework does not explicitly consider composition of softgoals or functional requirements. However, since in most cases, softgoals will affect each other, the complete Softgoal Interdependency Graph can provide good information about softgoal interdependencies. Some information about relationships of operationalisation decisions to functional requirements affected by them can also be obtained from the SIG. Yet, the complete picture of the system requirements is not available.

KAOS does not directly address composability. However, its formal representations maybe be reviewed with AND/OR operators and collecting the sub-goals into the super goal. KAOS also provides sets of heuristics (e.g., "do not overload the agent") which provide a perspective on the composite result of individual goals/tasks.

I* does not address the issues of composition. Yet, it is reasonable to expect that in many cases parts of agent's requirements could be modelled separately and require integration (e.g., due to system maintenance). In such cases it is essential to first integrate the agents: only the requirements of agents representing the same stakeholders can be composed, or the semantics of the analysis will be lost. This needs to be followed by the goal and softgoal interdependency graph integration for the merged agents: a problem that presently is not resolved.

In Problem Frames approach composite frames are built by joining simple frames through their common *domains*. Since each frame has its own description of a *domain* and PF does not have well defined composition semantics, except that in order to be composable the frames need to have the same domains. Presently, composition often requires solution-level information, thus departing from requirements to design and implementation issues. As a result, the composition is often an ad hoc process, and though Problem Frames have a potentially usable *joinpoint model* through the domains, these *joinpoints* are inconsistent between different frames. This approach does not allow quantification for composition either. Nevertheless, we should note that the most recent work on Problem Frames [25] has started to look at the issue of composition and a solution for cases with timed machine interference is suggested.

In Use Cases approach composition is achieved via *extend* and *include* relationships. The extension use cases are composed with the base use cases via *extend*; while repeatedly used sub-flows are composed with base flows via *include*. The composition of individual base use cases is not considered until design time, where still there is no explicit composition support, but mere merging of partial class information for each class from all use cases into a single representation of that object class in UML.

In respect with composition, Misuse Cases are quite similar to Use cases, except that additional relationships such as *mitigates, threatens, aggravates, conflicts with, has exception* allow to reflect more specific kinds of integration links.

AORE with Arcade supports requirement composability through a clear joinpoint model (where requirements in a viewpoint are joinpoints) and well defined composition semantics provided through its composition rules and operators. Moreover, the composition semantics are adaptable for each problem, as the set of composition operators is extensible.

In Aspects in Requirements Goal Models composition is carried out by matching the topics of the decomposed functional goals and the functional goal to which the non-functional task (so called *advising task*) applies (see Figure 3-29). The syntax of this composition still requires further research, as presently it is not well defined. Moreover, even the semantics of it are not quite clear.

AOSD/UC provides much more support for use case composition than the Use Case approach both at requirements level and later. As demonstrated in Figure 3-31 and Figure 3-34(c), a new construct for *extension pointcut* has been developed. It allows the provision of more details of extension use case application to base use cases. Examples of such details are condition of extension case application (e.g., as in Figure 4.19, 'when no room is available'), or the order of extension operation application to the use case (e.g., before, after, or instead of it). Also, the same extension case can be applied to many base use cases without having to clutter the use case diagram with notes, as it would be required with traditional Use Cases. Additionally, special design level composition semantics are provided.

The Scenario Modelling with Aspects approach has two alternative composition avenues for bringing together aspectual and non-aspectual scenarios: (a) composing interaction diagrams; (b) composing at state machine level. While (a) is more intuitive, it requires application of additional operators detailing how the messages from the aspectual and non-aspectual diagrams should interleave. This cannot be automated, as interleaving will depend on semantics of interaction. In (b), on the other hand, composition can be defined in terms of incoming and outgoing events, as well as states which are not available with option (a). In any case, manual mapping of roles to events, parameters (and states) is necessary. It should also be mentioned that the approach can easily accommodate composing functional scenarios together, while the semantics of composing non-functional scenarios together are not clear.

In the Aspectual Use Case Driven approach the composition of the Use Cases is extended with new use case relationships: *collaborate, damage,* and *constrain* [65]. Along with the standard *inherit, extend,* and *include*, these relationships help to incorporate both functional and non-functional use cases into the use case diagram. Besides, use of Use Case and Activity Pattern Specifications allow to use role parameters in use cases and activity diagrams. The composition requires a set of instantiation steps [65], where the Pattern Specification elements are replaced with concrete elements or other pattern elements that perform the necessary roles. However, this level of reuse and generality of use cases comes at the cost of one-to-one specification of every role binding for composition.

Concern Modelling with Cosmos does not directly address requirement or concern composition. Yet, the relationships that can be defined between concerns can be used to reflect how concerns should be treated together, e.g.,: does concern *A* require the presence of concern *B*, or should *A* be only included into the system if *B* is not there. This work provides no composition rules or operators or joinpoints, but allows all these to be specified as concerns of their own using the Cosmos schema.

The composability of CORE approach is the same as that of AORE with Arcade, discussed above. CORE, however, extends this composition model with the notion of concern projections and compositional intersection to facilitate extensive yet scalable analysis of interactions and trade-offs among concerns.

Requirement composition in AOREC differs from other composition approaches because here requirements are defined *per component*. Component requirements in AOREC are composed through the aspects: a component provides some aspect details as its requirement to an aspect and uses (requires) aspect details provides by other components. Thus, requirements are composed from the perspective of belonging to a component and contributing or using aspect details of an aspect. The aspect details are also clearly linked to the providing/using components via dashed arrows (which may turn unreadable for larger systems).

Theme/Doc in its *clipped action view* provides the order for theme composition, but it does not produce a view for composed requirements. Instead, it postpones actual composition to design level, where Theme/UML composition semantics are used. Hence, while Theme/Doc does have a clear *joinpoint model* (with requirements in themes acting as joinpoints) the *composition semantics* at requirements level are missing.

3.6.3 Evolvability

Table 3-4 shows the comparison of the features that support evolvability of requirements and change to their sources of origin for all the non-AO and AO approaches considered in section 0.

Approach	**Features That Support Evolvability**
PREview	not considered
VIM	tolerance for inconsistency, new template definitions, use of alternative representations, support for overlapping
NFRF	correlation catalogue; claims and augmentation on SIG, sub-goal decomposition; link of softgoal to functional goal
KAOS	thorough understanding of domain, detailed modelling of relations, reusable knowledge and requirements; sub-goal decomposition
I*	understanding reasons for requirements, use of NFR facilities
Problem Frames	sub-problem decomposition

Use Cases	use case decomposition; minimalist notation for use case diagrams
Misuse Cases	use case decomposition
AORE with Arcade	separation of concerns and composition concerns ; cross-reference tables
ARGM	correlation catalogue; claims and augmentation on SIG, sub-goal decomposition; link between functional goal-goal-task
AOSD/UC	use case decomposition; minimalist notation for use case diagrams; use case slices and modules
SMA	generic role elements; bindings
AUCDA	nfr templates, generic role elements, bindings
Cosmos	flexible schema; independence of concern representations
CORE	separation of concerns and composition concerns ; cross-reference tables; absence of base/aspect distinction
AOREC	use of aspect details
Theme/Doc	automation provided by the tool

Legend: PREview (section 3.3.1.1); VIM: Viewpoints and Inconsistency Management (section 3.3.1.2); NFRF: Non-Functional Requirements Framework (section 3.3.2.1); KAOS: (section 3.3.2.2); I* (section 3.3.2.3); PF: Problem Frames (section 3.3.3); Use Cases (section 3.3.4.1); Misuse Cases (section 3.3.4.2); AORE with Arcade: Aspect Oriented Requirements Engineering with Arcade (section 3.4.1.1); ARGM: Aspects in Requirements Goal Models (section 3.4.2.1); AOSD/UC: AOSD with Use Cases (section 3.4.3.1); SMA: Scenario Modelling with Aspects (section 3.4.3.2.1); AUCDA: Aspectual Use Case Driven Approach (section 3.4.3.3); Cosmos (section 3.4.4.1); CORE: Concern Oriented Requirements Engineering (section 3.4.4.2); AOREC: Aspect Oriented Requirements Engineering for Components (section 3.5.1.1); Theme/Doc (section 3.5.2.1).

Table 3-4: Summary of Features that Support Evolvability.

With regards to evolvability, in PREview removal or addition of a *concern* will result in heavy changes across all viewpoints requirements since the small number of *concerns* is assumed to be stable (a legacy of PREview's origin in the dependability domain).

From the perspective of evolvability the strongest part of VIM is its tolerance of inconsistency. Newly added or changed requirements artefacts may contain inconsistencies or conflicts with other artefacts, but these could be tolerated for a time or even for the life of the system, if the assessment of their removal cost is greater than that of risk from their retention. Besides, a new template with alternative representations can be defined for new viewpoints, if required. On the other hand, change to consistency rules or templates will result in a ripple effect of checks and changes through all artefacts.

The NFR framework initially considers the non-functional requirements and their decomposition one-by-one, before correlating them together in the SIG. Thus, the effect of a change will depend on its type. In the best case the change will affect only a single operationalisation or softgoal and have no influence on the rest of the SIG. This can happen if the change does not affect any major decisions and has no negative contributions to the existing softgoals. On the other hand, if the change requires re-consideration of major decisions or has large negative correlations with many other softgoals, the whole SIG could require re-evaluation. Also the correlation catalogue helps the developer to examine the cross-impact of the softgoals.

One of the main goals of KAOS is to facilitate knowledge reuse, which facilitates evolution. Reuse is achieved through cataloguing generalised knowledge, as well as domain-specific and instance knowledge. These can be reused by querying the KAOS knowledge base. Another important feature of KAOS is detailed analysis of both agents and their relationships all of which is recorded and can be easily applied to understand how to fit in the new/changed requirements or the effects of adding/removing an agent, etc. Also a change to a sub-goal, more often than not, can be localised to a given branch of the goal decomposition tree. However, the formal representation of KAOS artefacts while helpful in many cases, may result in added effort for understanding and maintenance.

Support for evolution is one of the goals of I* method which aims to prepare for the future change by understanding the reasons for present requirements. Having analysed the dependencies and the vulnerabilities of the stakeholders, their environment and volatile factors affecting it, this approach helps to make informed choices in system design that can anticipate the change. In addition this approach uses the NFR-style decomposition, thus benefiting from its correlation catalogues, claims and augmentations on SIG, and the sub-goal decomposition.

In PF, in some cases, a change in requirements can result in a change as serious as review of previously selected problem decomposition. However, most often a change may be reduced to a change in a particular sub-problem, without affecting other sub-problems. In any case, because there are no systematic composition means, all composition-related decisions will need to be reviewed [24].

Use cases are used to capture high-level user-centric requirements helping to structure required functionality and prioritise delivery of requirements. Thus, addition of new functionality can be treated as dealing with an extra use case, without affecting other use cases. At the same time, change of a use case functionality is also localised within that use case. All this suggests that use cases are well suited for tackling evolving requirements. Also, some guidelines are provided to assist in this, for instance [37] suggests to "never extend the extension" to avoid deep hierarchies of extend dependencies that could make use cases difficult to understand.

Though Misuse Cases are extension of the Use Cases approach, the evolvability of this approach is poorer due to relationships established between misuse cases and the main use cases. Any change in the main use cases will have to be also considered in terms of change of the currently related misuse cases and potential new misuse cases brought about by the change.

AORE with Arcade specifies each concern independently and provides a separate module for composition specification. Thus, change of a concern will affect only the evolving concern representation and the related composition specification for affected requirements. The approach uses a number of cross-reference tables to identify impacts and contributions of concerns. When a change occurs, in each of these tables the changed line/column will have to be reviewed, without affecting the rest of the tables. These tables also provide the necessary references to requirements whose composition should be reviewed. Evolution is also facilitated by the Arcade tool which can re-establish the potential trade-off points upon requirements change and evolution.

Similar to NFRF, Aspects in Requirements Goal Models could require reviews of the whole SIG in cases of significant requirement change. However, in most cases the main goals and softgoals of the system will be stable and changes will require reviewing only a branch or an operationalisation in a SIG. Additionally, due to the clear link between the functional and non-functional goals and tasks, the change in one can be clearly related to the others, using the composition specifications, as well as the graphs.

The discussion provided earlier for Use Cases, is also true for functional use cases in AOSD/UC, as in this respect there has been no major change made by AOSD/UC to the use cases approach. However, the addition of the infrastructure use cases for non-functional requirements will necessitate checks as to how the newly added or changed requirement in any use case affects these non-functional use cases. These might cause changes to be introduced to the non-functional use case specifications. On the other hand, use of use case slices may help in localising and locating use case related requirement artefacts.

Because Scenario Modelling with Aspects investigates scenarios, i.e. instances of interactions, change in a requirement will result in changes in one or more scenarios, causing the complete review of the affected scenarios, their interaction diagrams, binding and the composite scenario. However, due to separation of aspectual scenarios, a change in these will require review of elements only for that scenario, even if all its compositions with other non-aspectual scenarios will have to be reviewed. The review of scenarios due to change in requirements cannot be avoided, as scenarios are used precisely for understanding that behaviour. Thus, the merits of using separate aspectual scenarios will only be revealed in comparison with updating bindings and composed state machines with aspectual scenarios, as opposed to individually updating each full scenario and state machine if the aspectual part is not separated.

With the Aspectual Use Case Driven approach the modularity and localisation of change of functional concerns within a use case is preserved, as this approach is based on the Use Cases approach. For the non-functional concerns, any change requires review of the concern template's *Where* and *Requirements* sections in order to verify that the change does not invalidate the relationships and requirements of the NFR with the rest of the use cases. If changes have occurred in the way that NFR relates to the other use cases, the bindings of Use Case and Activity patterns to specific elements also need to be reviewed. On the other hand, the approach distinguishes between stable requirements and volatile ones [65]. The volatile ones – those prone to change – should map to Use Case Pattern Specifications and Activity Pattern Specifications and their change should result in composition review, but not review of any of the other use cases.

Concern Modelling with Cosmos is well suited for evolution as change will result in change of the concern and its related information (e.g., relationships, constraints if any) in the schema only, no other concern will be affected. Addition of a new concern simply entails recoding it along with its related information into the schema.

The evolvability of the CORE approach is similar to that of AORE, but is improved due to absence of a pre-specified "base and crosscutting" restriction. Thus, a concern initially identified as being non-crosscutting, can become crosscutting during the course of evolution. Such change will be accommodated in the model without major reworking of the requirements, except for corresponding cells in the relationship and contributions tables and the compositions.

AOREC does not assist with actual requirement evolution, but use of aspect details in AOREC clearly demarks the relationships between components, and so facilitates planning for requirement evolution. For instance, when there is a change in a component requirement, the impact can be evaluated in terms of effect on the provided or used aspect details. Questions such as, for instance,: is the changed aspect detail required by another component; is the newly required detail provided by others, will help in estimating the scale and cost of change.

Evolution in Theme/Doc is assisted by the automation provided by the tool. When a requirement is added or changed, it only requires re-generation of the views (using changed/added action words and entities) involving decision as to which theme does the new requirement belong to, and how the change/addition of a theme affects composition ordering. While the general composition order might be changed significantly, the themes themselves will be unaffected, except for the one that has been changed.

3.6.4 Scalability

Table 3-5 shows the comparison of the features that support scalability of requirements and change to their sources of origin for all the non-AO and AO approaches considered in section 0.

Approach	**Features That Support Scalability**
PREview	not considered
VIM	modularity of templates and viewpoint check rules
NFRF	catalogues for correlations and decompositions; records of decisions via claims
KAOS	catalogues for reusable knowledge, tactics and heuristics
I*	indirectly: NFR framework support through catalogues for correlations and decompositions; records of decisions via claims
Problem Frames	modularity of frames
Use Cases	use of coarse-grained use cases
Misuse Cases	use of coarse-grained use and misuse cases, per use case decomposition
AORE with Arcade	XML representation of artefacts and composition, extensibility of composition operators/actions set
ARGM	catalogues for correlations and decompositions; records of decisions via claims
AOSD/UC	use of coarse-grained use cases; packaging support
SMA	scalable state machine generating algorithm
AUCDA	use of coarse-grained use cases; partial use case diagrams
Cosmos	scalable schema; simple concern representation format
CORE	XML representation of artefacts and composition, extensibility of composition operators/actions set; use of XPath queries
AOREC	aggregate aspects , aspects
Theme/Doc	Coarse-grained actions, action groups

Legend: PREview (section 3.3.1.1); VIM: Viewpoints and Inconsistency Management (section 3.3.1.2); NFRF: Non-Functional Requirements Framework (section 3.3.2.1); KAOS: (section 3.3.2.2); I* (section 3.3.2.3); PF: Problem Frames (section 3.3.3); Use Cases (section 3.3.4.1); Misuse Cases (section 3.3.4.2); AORE with Arcade: Aspect Oriented Requirements Engineering with Arcade (section 3.4.1.1); ARGM: Aspects in Requirements Goal Models (section 3.4.2.1); AOSD/UC: AOSD with Use Cases (section 3.4.3.1); SMA: Scenario Modelling with Aspects (section 3.4.3.2.1); AUCDA: Aspectual Use Case Driven Approach (section 3.4.3.3); Cosmos (section 3.4.4.1); CORE: Concern Oriented Requirements Engineering (section 3.4.4.2); AOREC: Aspect Oriented Requirements Engineering for Components (section 3.5.1.1); Theme/Doc (section 3.5.2.1).

Table 3-5: Summary of Features that Support Scalability.

In PREview scalability is affected by two factors:

1. larger problems require more viewpoints and concerns to be identified to adequately cover the problem requirements;

2. the larger the number of identified viewpoints and concerns, the more likely that there will be conflicts between requirements, imposing need for negotiation, trade-offs and information management.

Since the approach does not provide sufficient support for conflict resolution and information management, the number of concerns usable per project is limited to about 6.

On one hand the modularity of VIM templates and the possibility of incremental provision of viewpoint consistency checking rules contributes to evolvability of the approach, but, on the other hand, it adds to the complexity of rules and to the need for additional checks.

The scalability of NFR SIG is limited: even for small examples it becomes cumbersome. The level of detail for each decision with supporting arguments and decomposition (such as including the attributes) makes the graph unmanageably large. Even with tool support this causes inconvenience. On the other hand, use of catalogues for correlations and decompositions and records of decisions via claims may assist in managing growing systems.

KAOS scalability may be hampered by the need to formalise each task, define pre and post conditions for each action and check that each action is performed as expected. On the other hand, the vast amount of knowledge and tactics and heuristics collected in the KAOS knowledge base is invaluable in supporting scalability by providing solutions to many potential problems.

Scalability is a bottleneck in the I* approach: even with tool support, working with large Strategic Rationale graphs is difficult. Use of NFR catalogues and claims would only weakly assist with some issues of scalability.

Problem Frames require a separate problem frame to be defined for each requirement with its corresponding frame concern. The number of graphs produced will increase linearly with the number of requirements. However, the biggest difficulty for scalability of this approach is caused by generation of the composite problem frame, when all individual frames and their concerns will need to be combined. Combining the graphs on a case-by case basis, where each new addition could require design or implementation specific knowledge, can become quite difficult. The feature that may support in scaling the system is modularity of each frame.

With the Use Cases approach each use case specifies a unit of useful functionality that the system provides to its actors. Thus, when new functionality is required, a new use case is added to the use case model, and though the model is graphical, each use case requires only a few additions, leading to a relatively good scalability for a graphical representation. However, for larger systems the developer faces a trade-off between dealing with too many use cases that overcrowd the use case diagram, or fewer use cases each of which is a larger and more complex unit. If the developers fall into the trap of 'functional decomposition' representing each function as a use case, the diagram will become cluttered and cumbersome.

Though misuse cases can benefit in terms of scalability from using coarse-grained cases, in general they give much poorer scalability than that of Use Cases. This is caused both because of a very large number of possible misuses to a use case, and due to the multiplication of their relationship links. These two factors together make even a small use case diagram rather unreadable. A partial solution is to consider each use case with its misuses separately, but this results in disjoined use cases and loss of the complete use case diagram.

The XML-based requirement representation and composition in AORE is very scalable. The only bottleneck of the approach for scalability could arise due to the tables used for correlation representation which can grow inconveniently large. Yet, this is simple enough to deal with using already existing everyday software tools, such as Excel tables, where the columns and rows irrelevant to the task in hand can be hidden, or different (e.g., reduced) views of the same table can be created with changes in one table reflected in all related ones.

The scalability problem of Aspects in Requirements Goal Models approach is even more severe then that of the NFRF approach. This is due to the direct merging of two SIGs (for functional and non-functional goal decomposition) into one. Although [41] suggests that advising parts of the graphs can be separated for analysis, it does not provide an adequate solution for managing oversize SIGs.

AOSD/UC faces the same scalability issue as discussed for Use Cases above. AOSD/UC also provides packaging support, whereby in large systems use case modules can be partitioned into use case packages. Each package can contain a set of use cases for a group of actors or package use cases according to the entities that the use case manipulates. Each such use case package should have a use case diagram to show use cases contained in it from a glance. Maintenance of separate packages is possible due to the composition support in the approach as opposed to the traditional Use Cases approach.

In the Scenario Modelling with Aspects approach scalability is hampered by the need to provide individual binding specifications not only for each aspect but also for each composition of that aspect separately. On the other hand, the scalability of the state machine generation algorithm has been shown to be satisfactory [63].

The scalability of Aspectual Use Case Driven approach is hampered with the problem of use case explosion (as for the Use Cases approach), augmented by the need to add a new use case and (particularly) its relationships for each NFR. The need to deal with relationships of several NFRs with the rest of the use cases will make the use case diagram unreadable. A possible partial solution is to build partial use case diagrams for each NFR, reflecting only the sets of use cases affected by a given NFR [47]. However, this solution will result in multiple use case diagrams for one system and will not depict the relationships between non-functional use cases themselves.

Concern Modelling with Cosmos is very scalable: any increase in the size of the project will only increase the number of modelled concerns. Nevertheless, though the Cosmos tool is scalable, the effort required for collection of the sufficiently complete set of concerns for a larger problem can be quite big due to the very large number of concerns involved [48].

The scalability of CORE approach is similar to that of AORE. In addition, XPath queries can be used to select specific concern projections and their cumulative effects.

The aggregate aspects in AOREC allow modularisation of requirements that affect groups of components. Thus, a coarse-grained relationship between aspect and a group is obtained, reducing repetitive specifications. Aspects too assist with supporting focus of component contributions/requirements on one concern at a time. On the other hand, the graphical representation of component and aspect requirements is not scalable, and easily becomes unreadable for larger systems.

Presently scalability is an unresolved issue for Theme/Doc: the graphical representation of actions and requirements in Theme/Doc tool become unmanageably large for even a medium size problem. Some possible solutions, such as ‘zooming out’ to a higher level of granularity, are currently under investigation [73].

3.6.5 Trade-off Analysis and Decisions Support

Table 3-6 shows the comparison of the features that support trade-off and decision support for all the non-AO and AO approaches considered in section 0.

Approach	**Features That Support Trade-Off Analysis And Decisions**
PREview	Rule “organisational concerns take precedence over viewpoint requirements”
VIM	cost/benefit analysis, inter-viewpoint consistency rules
NFRF	correlation catalogues; priority assignment to softgoals; SIG augmented with claims about past decisions and contribution types
KAOS	knowledge base, tactics, heuristics, priority assignment to goals
I*	Strategic Dependency and Strategic Rationale, use of NFR features
Problem Frames	partial support for assigning priorities to machines and events
Use Cases	not considered
Misuse Cases	not considered
AORE with Arcade	conflict detection through composition, contribution tables, temporal logic assertions of PROBE framework; weights assignment; stakeholder negotiations
ARGM	Conflict resolution process with Rule: “remove negative contribution link“
AOSD/UC	*before, after,* and *around* keywords for extension use cases
SMA	not considered
AUCDA	not considered
Cosmos	decision recording via constraints; concern relationships
CORE	conflict detection through composition, contribution tables, weights assignment; stakeholder negotiations, use of cumulative effects through folded tables and compositional intersection
AOREC	aspect details
Theme/Doc	not considered

Legend: PREview (section 3.3.1.1); VIM: Viewpoints and Inconsistency Management (section 3.3.1.2); NFRF: Non-Functional Requirements Framework (section 3.3.2.1); KAOS: (section 3.3.2.2); I* (section 3.3.2.3); PF: Problem Frames (section 3.3.3); Use Cases (section 3.3.4.1); Misuse Cases (section 3.3.4.2); AORE with Arcade: Aspect Oriented Requirements Engineering with Arcade (section 3.4.1.1); ARGM: Aspects in Requirements Goal Models (section 3.4.2.1); AOSD/UC: AOSD with Use Cases (section 3.4.3.1); SMA: Scenario Modelling with Aspects (section 3.4.3.2.1); AUCDA: Aspectual Use Case Driven Approach (section 3.4.3.3); Cosmos (section 3.4.4.1); CORE: Concern Oriented Requirements Engineering (section 3.4.4.2); AOREC: Aspect Oriented Requirements Engineering for Components (section 3.5.1.1); Theme/Doc (section 3.5.2.1).

Table 3-6: Summary of Features that Support Trade-Off Analysis and Decisions.

In PREview the need for trade-off and conflict resolution is identified when viewpoints and concerns display negative relations in decision tables. PREview does not provide any particular trade-off resolution mechanism, but simply acknowledges the need for it.

The only direction given by this approach is that the organisational concerns take precedence over viewpoint requirements.

In VIM identification of trade-off points in case of inconsistency is assisted by the consistency checking rules, but these rules may not detect conflicts where the goals of stakeholder differ. When an inconsistency or conflict is detected, the cost-benefit analysis of the alternative handling strategies are considered (e.g., eliminate or ignore?) and the most suitable strategy selected. VIM does not detail how to detect conflicts or how to carry out cost-benefit analysis: these are left open for the method used to decide.

NFRF provides good support for identification of trade-off points via correlation catalogues which help the developer to examine the cross-impact of the softgoals and decide between competing alternative solutions. The approach allows priority assignment to softgoals (by stakeholders or developers). The NFR SIG helps to visually pass over alternative choices, see their interrelationships and influences and make an informed decision. NFR also provides good support for decision recording via claims and augmentations as well as contribution types (e.g., weak positive, strong negative, etc.).

KAOS allows priority values (between 0 and 1) to be assigned to goals. These are used during conflict resolution or alternative selection for decomposition: the goals with higher priorities are favoured. Decision support in KAOS is provided via reusable knowledge of the knowledge base and also through sets of tactics and heuristics designed to support the KAOS process in case of difficulties.

I* assists in resolving trade-offs from the process level to that of an individual task. The Strategic Dependency and Rationale graphs allow to model and evaluate alternative channels of stakeholder interest satisfaction. The principle of "reciprocal dependency" [21] (which suggests that in order to ensure dependum delivery, there should exist a reciprocal dependence between dependee and depender) helps in checking the viability of alternatives. The NFR contribution types help in assessing the degree of contributions. Weights and priorities may also be used to decide between alternatives.

The main work on Problem Frames [23] did not consider conflict resolution and trade-off support. More recent work in [25] has begun to look at conflict resolution, and addresses a certain set of such problems by assigning priorities to machines, events, etc. This, however, is not a sufficiently complete solution, as it addresses only a small subset of problems where timed machine interference can be used.

The Use Cases approach does not provide any conflict resolution because it simply records all user requirements in separate use cases and does not attempt any reconciliation. This is left to be addressed at the design time, when the user requirements are mapped to object classes.

Misuse Cases provide numerous misuse possibilities per use case. The requirements engineer does not always accept all the possible cases, but selects the ones s/he considers relevant. Nevertheless, this approach does not provide any guidelines, heuristics, or any other support for misuse case selection decisions. Similar to Use Cases, this approach also does not provide any conflict resolution support.

AORE with Arcade provides good support for trade-off analysis: the need for these is initially detected through the composition process of requirements, contribution tables, as well as through temporal logic assertions of the PROBE framework at the later stages of development. Once detected, the conflicting requirements are evaluated against the weights assigned to them by the stakeholders, the requirements with lesser weights are weakened to resolve the conflict (with approval from stakeholders). However, if these have equal weights, the stakeholders are invited for negotiations.

In the Aspects in the Requirements Goal Models approach, when conflicts are detected between the SIG nodes (i.e., goals, softgoals, or tasks) the contribution link from the negatively contributing node to the parent node is simply removed. While this approach does remove the conflicts, it is in danger of losing relevant links between goals and tasks. It also assumes that a goal can be achieved through means where conflicting solutions can be ignored. This may not always be possible, or sometimes may result in a sub-standard solution compared to those where conflicts are actually resolved rather than removed at the source.

Similar to Use Cases, AOSD/UC does not attend to conflict resolution at the requirements engineering level. The only addition of AOSD/UC at requirements level, that could address some ordering of conflicts, is that of *before, after,* and *around* keywords which clarify the order of operation application of the extension use cases to the base ones.

Scenario Modelling with Aspects does not provide any support for trade-off analysis, though this could often be required when defining bindings of aspectual scenarios to specific elements in non-aspectual ones.

Presently the Aspectual Use Case Driven approach does not provide any trade-off resolution support either.

Concern Modelling with Cosmos does not need to resolve trade-offs between concerns for concern modelling. However, any trade-off decisions taken during development can be recorded as constraints in the schema for the relevant concerns. The concern relationships are a helpful source of information when making such decisions, as they help to understand the implications of such decisions on the concerns involved.

The trade-off support in CORE is mainly similar to that of the AORE with Arcade. Additionally, the use of cumulative effect of concerns on each other can help in more informed decision making.

AOREC does not provide any explicit trade-off analysis or decision support, except for the possibility to evaluate the impact of a selected decision on provided/required aspect details and their effects on other components.

Theme/Doc does not provide any explicit support for conflict identification and resolution or trade-off decision making. Though some trade-offs are required even when applying the method (e.g., when deciding on major/minor actions, or base/secondary role of the action for the requirement) these are made implicitly, using the developer's experience and intuition.

3.6.6 Support for Mapping

Table 3-7 shows the comparison of the features that support mapping of requirements to types of artefacts (e.g., decisions, structures, procedures, etc.) of later stages of lifecycle for all the non-AO and AO approaches considered in section 0.

Approach	**Features That Support Mapping**
PREview	templates for concern decomposition

VIM	work plan viewpoint trigger actions, inter viewpoint check rules
NFRF	type and decomposition catalogues; operationalisations – functional requirement links; contribution records
KAOS	decomposition structures of the knowledge base; action-to-agent assignments
I*	actors, use of NFR framework (type and decomposition catalogues; contribution records) and goal and softgoal operationalisations
Problem Frames	not considered
Use Cases	collaboration diagrams, some guidelines
Misuse Cases	collaboration diagrams, some guidelines
AORE with Arcade	Guidelines
ARGM	type and decomposition catalogues; operationalisations – functional requirement links; contribution records
AOSD/UC	collaboration diagrams, some guidelines; pointcuts and aspects
SMA	not considered
AUCDA	collaboration diagrams, some guidelines
Cosmos	concern relationships; allows recording of mapping decisions
CORE	Guidelines
AOREC	use of same aspects and details at both requirements and design levels.
Theme/Doc	closeness of requirements and design models; *theme view* of the Theme/Doc tool

Legend: PREview (section 3.3.1.1); VIM: Viewpoints and Inconsistency Management (section 3.3.1.2); NFRF: Non-Functional Requirements Framework (section 3.3.2.1); KAOS: (section 3.3.2.2); I* (section 3.3.2.3); PF: Problem Frames (section 3.3.3); Use Cases (section 3.3.4.1); Misuse Cases (section 3.3.4.2); AORE with Arcade: Aspect Oriented Requirements Engineering with Arcade (section 3.4.1.1); ARGM: Aspects in Requirements Goal Models (section 3.4.2.1); AOSD/UC: AOSD with Use Cases (section 3.4.3.1); SMA: Scenario Modelling with Aspects (section 3.4.3.2.1); AUCDA: Aspectual Use Case Driven Approach (section 3.4.3.3); Cosmos (section 3.4.4.1); CORE: Concern Oriented Requirements Engineering (section 3.4.4.2); AOREC: Aspect Oriented Requirements Engineering for Components (section 3.5.1.1); Theme/Doc (section 3.5.2.1).

Table 3-7: Summary of Features that Support Mapping.

In PREview mapping of concerns to requirements is supported through templates for concern decomposition. However, mapping of requirements to the artefacts of later lifecycle stages is not supported, barring a very weak proposition to derive the high level architecture alongside software requirements specification development.

In VIM, the inter viewpoint checking rules developed by the method users may also include rules on how to map specific artefacts to their representations in other notations. These rules may be required to be checked for a viewpoint template instantiation, thus supporting artefact mapping. On the other hand such rules and related automation support is not readily available and will have to be developed by the users of the methodology.

The NFRF type and decomposition catalogues assist in mapping concerns to non-functional requirements. The links between operationalisations and target system's functional requirements also help to implicitly envisage their mapping to the design artefacts. Also the contribution records inform and helps to make decisions on particular decompositions and operationalisations. Thus, the major architectural choices and design decisions about the non-functional requirements can be taken along with the softgoal decomposition process, though no specific guidelines for mapping (other than the SIG construction guidelines) are provided.

KAOS has decomposition structures similar to those of NFRF decomposition catalogues. These too help in mapping non-functional, but also functional, goals onto functionalities or tasks that can be directly implemented. Some tasks (e.g., for the non-functional goals) may map to procedures which will not be automated by the software system, but may become processes or procedures in the wider environment of the system, etc. On the other hand, assignments of actions to agents can also be perceived as mapping of the functionality onto a potential class.

I* may help in identification of major architectural modules via agent and top-level goal identification, yet this approach does not detail how the Strategic Dependency and Rationale graphs should map to architecture. On the other hand, by using the NFR-style decomposition of its goals and softgoals, the approach results in specific decomposition and operationalisation choice commitment, as discussed for NFRF above.

The Problem Frames approach does not provide any significant support for mapping frames onto later stages of development, though in some cases it relies on knowledge of later stage artefacts for decisions on frame composition or modelling some phenomena [75].

Use Cases provide a good mapping support for requirements-level use cases to their design realisations through collaboration diagrams – some guidelines are also provided to this end. However, there is no support for mapping crosscutting functional requirements onto separate units (the crosscutting non-functional ones are not addressed in this approach at all). Also, it should be noted that the collaboration diagrams do not represent the final design artefacts for the use cases: to conform to OO design, all corresponding partial class representations from the collaborations will have to be combined into a single design class for each OO class representation.

Unlike a traditional use case, a misuse case will not always map to a design-level use case. Often a misuse case will result in a decision on the business process, or procedure or alike. Yet, some misuse cases will indeed become design-level use cases. In this respect, the misuse cases approach does not provide any mapping support or distinction for misuse cases. Regarding the use cases part of the misuse cases approach, the discussion is the same as that for Use Cases.

AORE with Arcade provides guidelines for mapping requirements to later stages of software development (referred to as *aspect dimension specification* in the approach). However, presently these guidelines are mainly intuitive, and need to be documented more effectively and fully.

The Aspects in Requirements Goal Models is completely similar to NFRF in providing mapping support.

Similarly to Use Cases approach, AOSD/UC supports mapping of functional use cases to designs through collaboration diagrams; but this time these representations can remain independent at the design time (due to the AOSD/UC defined design-level composition support). At the requirements engineering stage, the concepts of pointcuts and aspects are used to map the crosscutting relationships between base and extension use cases.

Scenario Modelling with Aspects does not consider scenario mapping to the later stages of software development.

Aspectual Use Case Driven approach does not provide any extra mapping support on top of that already available from the Use Cases approach for the functional use cases.

The Concern Modelling with Cosmos approach does not consider mapping the concerns onto any artefacts. That should be done within a particular development methodology employed for development. Nevertheless, Cosmos can help in recording the mapping to physical concerns (artefacts) and also provide information on the implications of mapping from the perspective of influence of other concerns.

The mapping support of CORE is same as for AORE. It also extends the mapping guidelines to map the requirements-level concerns and trade-off analysis to potential architectural choices.

In AOREC the same aspects and aspect details are used that were identified at the requirements stage. During design the requirements aspects are simply further elaborated.

The concerns grouped at the requirement engineering stage into a theme in the theme view of the Theme/Doc tool are neatly mapped to Theme/UML themes due to closeness of their requirements and design models.

3.6.7 Homogeneity of Concern Treatment

Legend: PREview (section 3.3.1.1); VIM: Viewpoints and Inconsistency Management (section 3.3.1.2); NFRF: Non-Functional Requirements Framework (section 3.3.2.1); KAOS: (section 3.3.2.2); I* (section 3.3.2.3); PF: Problem Frames (section 3.3.3); Use Cases (section 3.3.4.1); Misuse Cases (section 3.3.4.2); AORE with Arcade: Aspect Oriented Requirements Engineering with Arcade (section 3.4.1.1); ARGM: Aspects in Requirements Goal Models (section 3.4.2.1); AOSD/UC: AOSD with Use Cases (section 3.4.3.1); SMA: Scenario Modelling with Aspects (section 3.4.3.2.1); AUCDA: Aspectual Use Case Driven Approach (section 3.4.3.3); Cosmos (section 3.4.4.1); CORE: Concern Oriented Requirements Engineering (section 3.4.4.2); AOREC: Aspect Oriented Requirements Engineering for Components (section 3.5.1.1); Theme/Doc (section 3.5.2.1).

Table 3-1: Summary of Features that Support Traceability of Requirements and Change to their Sources of Origin Criterion.Table 3-8 shows the comparison of the features that support homogeneity of requirements treatment for all the non-AO and AO approaches considered in section 0.

Approach	**Features That Support Homogeneity of Concern Treatment**
PREview	n/a
VIM	n/a
NFRF	n/a
KAOS	equal importance of functional and non-functional goals, decomposition structures for both, single formal language representation
I*	equal importance of functional and non-functional goals, decomposition structure for both
Problem Frames	n/a
Use Cases	support representation of crosscutting and non-crosscutting functional concerns via *extend* and *include* relationships
Misuse Cases	support of the Use Cases approach, support for some non-functional concerns by countering possible misuses.
AORE with Arcade	"in step" identification and treatment of functional and non-functional, crosscutting and non-crosscutting concerns.
ARGM	equal importance of functional and non-functional goals, decomposition of both

	alongside each other.
AOSD/UC	support of the Use Cases approach, use case representation for functional and non-functional, crosscutting and non-crosscutting concerns
SMA	scenario modelling for functional and non-functional, crosscutting and non-crosscutting concerns
AUCDA	support of the Use Cases approach, use case representation for functional and non-functional, crosscutting and non-crosscutting concerns
Cosmos	single possible way of representing all concerns as *concerns* in general
CORE	single way of concern treatment, independent of concern type, accommodating all type of concerns
AOREC	n/a
Theme/Doc	single uniform treatment of all identified concerns, accommodating all types of concerns

Legend: PREview (section 3.3.1.1); VIM: Viewpoints and Inconsistency Management (section 3.3.1.2); NFRF: Non-Functional Requirements Framework (section 3.3.2.1); KAOS: (section 3.3.2.2); I* (section 3.3.2.3); PF: Problem Frames (section 3.3.3); Use Cases (section 3.3.4.1); Misuse Cases (section 3.3.4.2); AORE with Arcade: Aspect Oriented Requirements Engineering with Arcade (section 3.4.1.1); ARGM: Aspects in Requirements Goal Models (section 3.4.2.1); AOSD/UC: AOSD with Use Cases (section 3.4.3.1); SMA: Scenario Modelling with Aspects (section 3.4.3.2.1); AUCDA: Aspectual Use Case Driven Approach (section 3.4.3.3); Cosmos (section 3.4.4.1); CORE: Concern Oriented Requirements Engineering (section 3.4.4.2); AOREC: Aspect Oriented Requirements Engineering for Components (section 3.5.1.1); Theme/Doc (section 3.5.2.1).

Table 3-8: Summary of Features that Support Homogeneity of Concern Treatment

In PREview the organisational concerns (i.e. the crosscutting non-functional concerns) are considered of prime importance. When the functional concerns are elicited, it is ensured that the influence of the organisational concerns on the functionality is considered. If necessary, the functionality is adjusted to maintain the requirements of the organisational concerns. Crosscutting functional concerns have been overlooked entirely.

VIM does not address the issue of crosscutting concern treatment either for functional or non-functional concerns. In fact the non-functional concerns do not appear to be considered in general.

The concern treatment in NFRF is very similar to that in PREview: non-functional concerns are prioritised over functional ones and the crosscutting functional concerns are overlooked. However, unlike PREview, the effect of non-functional concerns on functionality is not analysed, though their effects on other non-functional concerns is detailed in the SIG.

In KAOS both *functional* and *non-functional* concerns (or goals) are treated via the same procedure: none is considered more important than the other. The mutual influences of the gals can also be studied by looking at the relationships of the objects and agents associated to a goal or its task, though the appropriate treatment of *crosscutting* concerns is not explicitly addressed.

In I* the functional concerns are represented as goals and tasks, and non-functional ones as softgoals. Both are decomposed to task and finally operationalisation levels, and so are treated similarly. However, I* does not separately address the issue of crosscutting and non-crosscutting concerns.

As already mentioned, Problem Frames do not treat non-functional concerns in any systematic way. While functionality is dealt with through problem frames, non-functional concerns are mainly discussed as "also concerns" and treated in an ad hoc way (except for reliability). Crosscutting concerns (either functional or non-functional) are not explicitly acknowledged either.

In the Use Cases approach the crosscutting functional and non-crosscutting functional concerns are treated quite evenly at the requirements engineering stage. The crosscutting functional use cases are expressed through *extend* and *include* and are elaborated along with functional use case identification. The non-functional concerns are not addressed by this approach.

Misuse cases are similar to use cases with regards to homogeneity. However, through the misuse cases some issues of non-functional concerns can also be addressed. For instance, security concern can be explored through this method.

In AORE with Arcade all concerns are treated evenly at all steps of the approach: viewpoints are the base decomposition, with organisational concerns crosscutting them, but they are identified and treated in step with each other. There is a base-aspect separation in that viewpoints act as the base for observing influences and trade-offs of aspects.

Unlike NFRF, the Aspects in Requirements Goal Models approach considers both functional (goals) and non-functional (softgoals) crosscutting concerns as part of goal-aspect building. Both types of goals are decomposed alongside and their corresponding operationalisations chosen in step with decomposition. Thus, this approach treats all concerns quite evenly.

Both functional and non-functional concerns in AOSD/UC are treated via use cases. However, while functional concerns are use cases on their own, non-functional concerns are treated as extensions to a specific Perform Transaction use case, thus becoming "lesser" use cases then the functional ones. Nevertheless, at the requirements engineering stage, all kinds of concerns appear to be given sufficient regard.

Scenario Modelling with Aspects is well suited for representing and treating both functional and non-functional crosscutting and non-crosscutting concerns, which can be represented by scenarios. Yet, as has been mentioned earlier, it is not clear how fully the non-functional concerns can be covered by scenarios.

Aspectual Use Case Driven approach provides equal attention to both functional and non-functional crosscutting concerns. Both of these types are represented in the use case diagram and activity diagrams. The issues related to identification and treatment of both concern types are of equal importance in this approach.

In Concern Modelling with Cosmos all concerns are treated similarly: as concerns in general. The only distinction between them is their mapping to different types within the Cosmos schema. Presently in this approach concern identification is decided manually by the requirements engineer.

CORE has a homogeneous concern treatment process: it does not make any distinctions between functional or non-functional, crosscutting or non-crosscutting concerns.

During AOREC the functional concerns are assigned to particular components, while non-functional ones are assigned to aspects. While there are no clear instructions for this, it is generally expected that the functional assignment will be carried out first, then aspects identified. There also has been no example of using aspects for functional crosscutting concerns. Thus, we conclude that requirements allocated per component are the base decomposition elements, while aspects are an additional crosscutting dimension for non-functional requirements.

Theme/Doc's methodology of concern treatment through graphs could be applicable to both functional and non-functional crosscutting concerns. The unresolved problem, however, lies in identification of non-functional concerns in absence of actions associated with them, and even more importantly, ensuring that this "absent action" is identified in *all* affected requirements.

3.6.8 Verification and Validation

Legend: PREview (section 3.3.1.1); VIM: Viewpoints and Inconsistency Management (section 3.3.1.2); NFRF: Non-Functional Requirements Framework (section 3.3.2.1); KAOS: (section 3.3.2.2); I* (section 3.3.2.3); PF: Problem Frames (section 3.3.3); Use Cases (section 3.3.4.1); Misuse Cases (section 3.3.4.2); AORE with Arcade: Aspect Oriented Requirements Engineering with Arcade (section 3.4.1.1); ARGM: Aspects in Requirements Goal Models (section 3.4.2.1); AOSD/UC: AOSD with Use Cases (section 3.4.3.1); SMA: Scenario Modelling with Aspects (section 3.4.3.2.1); AUCDA: Aspectual Use Case Driven Approach (section 3.4.3.3); Cosmos (section 3.4.4.1); CORE: Concern Oriented Requirements Engineering (section 3.4.4.2); AOREC: Aspect Oriented Requirements Engineering for Components (section 3.5.1.1); Theme/Doc (section 3.5.2.1).

Table 3-1: Summary of Features that Support Traceability of Requirements and Change to their Sources of Origin Criterion.Table 3-9 shows the comparison of the features that support verification and validation of requirements with respect to their design and implementation outputs for all the non-AO and AO approaches considered in section 0.

Approach	**Features That Support Verification and Validation**
PREview	not considered
VIM	multiple representations, consistency rules
NFRF	"walking through" SIG
KAOS	formalism of the representation, "walk though" decomposition graph
I*	participatory development of the Strategic Dependency and Rationale graphs, walking through goal and softgoal interdependency graphs, scenario generation.
Problem Frames	"walking through" frame concern
Use Cases	simplicity of use case model, the text-based style of use case specifications, scenarios and interaction diagrams are all well suited for user validation; acceptance and black-box test descriptions can be derived from scenarios
Misuse Cases	the text-based style of use case specifications, scenarios and interaction diagrams for misuse validation; acceptance and black-box test descriptions derived from misuse case scenarios
AORE with Arcade	"walking through" the composed requirements; proof obligations from PROBE framework for model checking and test cases;
ARGM	"walking through" SIG
AOSD/UC	simplicity of use case model, the text-based style of use case specifications, scenarios and interaction diagrams are all well suited for user validation; acceptance and black-box test descriptions can be derived from scenarios
SMA	executable state machine specifications; simplicity of use case model, the text-based style of use case specifications, scenarios and interaction diagrams are all well suited for user validation; acceptance and black-box test descriptions can be derived from scenarios
AUCDA	partial projections of the functional use cases per an NFR use case; simplicity of use case model, the text-based style of use case specifications, scenarios and interaction diagrams are all well suited for user validation; acceptance and black-box test descriptions can be derived from scenarios

Cosmos	textual representation of artefacts
CORE	"walking through" the composed requirements
AOREC	matching provided/required aspect details per aspects
Theme/Doc	"walking through" the action view and clipped action view graphs; informal validation of designs against the themes

Legend: PREview (section 3.3.1.1); VIM: Viewpoints and Inconsistency Management (section 3.3.1.2); NFRF: Non-Functional Requirements Framework (section 3.3.2.1); KAOS: (section 3.3.2.2); I* (section 3.3.2.3); PF: Problem Frames (section 3.3.3); Use Cases (section 3.3.4.1); Misuse Cases (section 3.3.4.2); AORE with Arcade: Aspect Oriented Requirements Engineering with Arcade (section 3.4.1.1); ARGM: Aspects in Requirements Goal Models (section 3.4.2.1); AOSD/UC: AOSD with Use Cases (section 3.4.3.1); SMA: Scenario Modelling with Aspects (section 3.4.3.2.1); AUCDA: Aspectual Use Case Driven Approach (section 3.4.3.3); Cosmos (section 3.4.4.1); CORE: Concern Oriented Requirements Engineering (section 3.4.4.2); AOREC: Aspect Oriented Requirements Engineering for Components (section 3.5.1.1); Theme/Doc (section 3.5.2.1).

Table 3-9: Summary of Features that Support Verification and Validation.

PREview collects requirements from individual viewpoints. These requirements can often be merged or amended due to conflict resolution or influence of organisational concerns. Nevertheless, the approach does not provide any support for verification of the end-produced requirements against those from the individual viewpoints. Neither does it support validation of the derived architecture and designs against requirements.

The viewpoint check rules in VIM can be used to verify the consistency of different viewpoints, as well as their mappings to the different lifecycle stage artefacts, if such rules are specified in each given case. Verification of individual viewpoints too can be accomplished through participation of the agent from whom the viewpoint was elicited. However, these cannot ensure that the viewpoints are conflict free or that each viewpoint representation will be understood by the source stakeholder (e.g., if defined in a formal representation, etc.).

NFRF does not provide specific verification and validation procedures either, though its SIG can be used for "walking through" the concern decomposition and its verification with the user (if appropriate).

The formalism of KAOS artefacts allows precise analysis to be carried out for the system artefacts, thus supporting verification. On the other hand, the gaol decomposition graph is quite informal and can be used for "walk through" validation with the users.

One of the attractions of the I* approach is its intuitiveness and ease of stakeholder integration into dependency and rationale graph development. Through the "participatory" involvement of the stakeholders, the software developers can achieve a verifiable and reliable model for the system development. These graphs can also be used for verification via scenario generation (e.g., what if agent x did not deliver on dependum y?). In addition, the NFRF-style interdependency graph can be "walked through" to verify specific goal/task decomposition and operationalisation.

In the Problem Frames approach the composite frame concern is an excellent way of "walking through" and verifying the problem and its solution with the users to ensure that they agree with assumed domain properties and are satisfied with the way that the requirements will be met by the system. The formal validation of designs and implementation against the requirements is not discussed in this work.

With the Use Cases approach the simplicity of use case model, the text-based style of use case specifications, scenarios and interaction diagrams are all well suited for user validation. Use cases also help to explore various ways in which a system is used and derive acceptance and black-box test descriptions from scenarios against which the final system can be verified. In essence, each scenario in a use case is a test case ready for validation.

For verification and validation the Misuse cases are used in the same way as Use Cases.

Verification in AORE with Arcade can be easily accommodated by "walking through" the composed requirements (which will appear in text due to their XML representation) with the customers. Validation, on the other hand is thoroughly supported through the PROBE framework which provides proof obligations for aspectual requirements and associated trade-offs. The proof obligations generated in PROBE can be used as input to both formal method tools (such as model checkers) or as a basis to derive test cases.

Aspects in Requirements Goal Models approach does not provide specific verification and validation procedures, though the SIG can be "walked through" for verification purposes and the discussion for NFR framework is also applicable here.

From the verification and validation perspective at requirements engineering level AOSD/UC fits the same description as Use Cases approach, discussed above.

The final aim of Scenario Modelling with Aspects approach is to generate executable state machine specifications to allow users to validate the requirements by injecting events into the modelled system, thus making it highly suitable for validation. Besides, the use cases and scenarios can also be verified with the users, as with the Use Cases approach.

In the Aspectual Use Case Driven approach verification and validation of functional use cases is same as for the Use Cases approach. Additionally, the non-functional requirements and their effect on the functional use cases can be validated with the partial projections of the functional use cases per NFR use case. Yet, the influences of non-functional concerns on each other are not verifiable and their validation is not considered either.

Concern Modelling with Cosmos could be helpful in validating the artefacts and their relationships against original concerns and their relationships, as well as any defined constraints. Concerns and relationships themselves can be discussed with the users, to verify their correctness; the textual representation of artefacts can be helpful in this respect. However, presently Cosmos does not provide any formal verification and validation support.

Verification in CORE is same as that of AORE. The validation process may also become quite similar once the PROBE framework is adapted to the multi dimensional separation of concerns used in CORE. This adaptation is intended to be performed in the near future.

AOREC has no explicit verification and validation support either for verifying requirement assignment per component, or aspect detail assignment per aspect. On the other hand, the aspect details can be used to verify if the details required by a component are provided by another component, thus checking the correctness of component composition.

Verification with the customers by "walking through" the action view and clipped action view graphs is possible for Them/Doc. Informal validation of designs against the themes produced at the requirements stage is also possible. Though this validation cannot prove correctness of designs and will not provide a 1-1 mapping between requirements themes and design operations, it helps to put the design decisions in context of their corresponding requirements and verify the decisions themselves [55].

4. AO Architecture

4.1 Introduction: Architecture

Software architectures are high-level design representations that facilitate the communication between different stakeholders, enable the effective partitioning and parallel development of the software system, provide a means for directing design decisions and their evaluation, and finally provide opportunities for reuse [76] [77].

Software architecture is generally considered to play a fundamental role in coping with the inherent difficulties of the development of large-scale and complex software systems [78]. A common assumption is that architecture design should support the required software system qualities such as robustness, adaptability, reusability and maintainability [79] [76].

The term architecture is not new and has been used for centuries to denote the physical structure. A common definition that is applied to the context of software systems is the following [76]:

> "The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them".

Software architecture forms one of the key artefacts in the entire software development life cycle since it embodies the earliest design decisions and includes the gross-level components that directly impact the subsequent analysis, design and implementation. Accordingly, it is important that the architecture design supports the software system qualities required by the various stakeholders. For ensuring the quality factors it is necessary to identify the fundamental concerns for architecture design and various architecture design methods have been introduced for this purpose.

Current software architecture design methods, however, do not make an explicit distinction between conventional architectural concerns that can be localised using current architectural abstractions and architectural concerns that crosscut multiple architectural components. The risk is that potential aspects might be easily overlooked during the software architecture design and remains unsolved at the design and programming level. This may lead to tangled code in the system and consequently the quality factors that the architecture analysis methods attempt to verify will still be impeded. Similar to the notion of aspect at the programming level, these concerns are crosscutting and denote so-called architectural aspects. Since the crosscutting property of architectural aspects is inherent, these cannot be undone simply by redefining the software architecture using conventional architectural abstractions. In fact, like various aspect-oriented programming abstractions, we need explicit mechanisms to identify, specify and evaluate aspects at the architecture design level. In this sense aspectual architecture design approaches describe steps for identifying architectural aspects and their related tangled components. This information is used to redesign the given architecture in which the architectural aspects are made explicit. This is different from traditional approaches where architectural aspects are implicit information in the specification of the architecture.

The survey of architecture approaches in this report discusses state-of-the-art in both non-AO and AO architecture approaches from the perspective of treating crosscutting concerns at the architecture level. Each set of approaches (i.e., AO and non-AO) is categorised as follows:

1. *Approaches for modelling architecture* - include all the approaches for modelling architectures visually or textually.
2. *Approaches for architecture design process* - include the approaches that provide explicit process and heuristic rules for designing architectures
3. *Architecture evaluation approaches* - include the approaches that mainly focus on the analysis of architecture with respect to required quality criteria

We use the general quality criteria from section 2 as a basis to provide a qualitative comparison between the various approaches.

This section is further organised as follows: section 4.2 describes the non-AO approaches, section 4.3 presents the AO approaches, section 4.4 discusses the comparison.

4.2 Non-AO Approaches

4.2.1 Architectural Modelling Approaches

This section discusses the architectural modelling approaches which are grouped into:

- Architectural Description Languages or ADLs (section 4.2.1.1), represented by ACME [80], Chiron-2 [81, 82], Aesop [83, 84], Darwin [85, 86], Rapide [87, 88], and Wright [89] approaches;
- Approaches that use Unified Modelling Language (UML) for software architecture modelling (section 4.2.1.2), represented by [90];
- Architecture Evaluation Methods (section 4.2.3), represented by Software Architecture Analysis Method [91] (SAAM), Architecture Trade-Off Analysis Method [92], SAAM Founded on Complex Scenarios [93], Extending SAAM by Integration in the Domain [94], SAAM Evolution and Reusability [95], and Architecture Level Prediction of Software Maintenance [96] approaches.

4.2.1.1 Architecture Description Languages

Several ADLs have been proposed as modelling representations to support architecture-based development. In general, an ADL is used to model the components, the connections, and the configuration of software architecture. The set of ADLs in the literature is quite broad and several surveys have already been published by various authors, e.g., [97, 98]. Although each ADL provides a specification language for representing the high level structure of the system we can identify several differences. First of all, it appears that several ADLs focus on modelling particular domains [99] whereas other ADLs are general-purpose. Secondly, different ADLs focus on specification of different characteristics of the architecture. Some focus on the specification of the architectural components, some on interaction of the components and others on the configuration. Finally, ADLs can be distinguished by their goals. While most of them can be considered as a means for communication and understanding, some of them can be considered as models that can be adopted to derive the subsequent artefacts from it.

Since the set of ADLs is too broad to consider, we will not redo the work on classifying ADLs but suffice to refer to the surveys [97, 98] and, in addition, describe the ADLs that are representative of the above distinctions. The representative ADLs discussed below are ACME [80], Chiron-2 [81, 82], Aesop [83, 84], Darwin [85, 86], Rapide [87, 88], and Wright [89].

4.2.1.1.1 ACME

ACME [80] builds on the experience of other ADLs and intends to serve as a common representation for software architectures. ACME supports the definition of software architectures from four distinct perspectives:

1. Structuring of a system into its constituent parts;
2. Properties of interest about a system or its parts that allow one to reason abstractly about overall behaviour (both functional and non-functional);
3. Constraints on how the architecture can change over time;
4. Types and styles defining classes and families of architecture.

ACME focuses on architectural interchange, predominantly at the structural level [98]. An architecture description in ACME can comprise of seven types of entities: *components*, *connectors*, *systems*, *ports*, *roles*, *representations*, and *rep-maps*. The first five are illustrated in Figure 4-1.

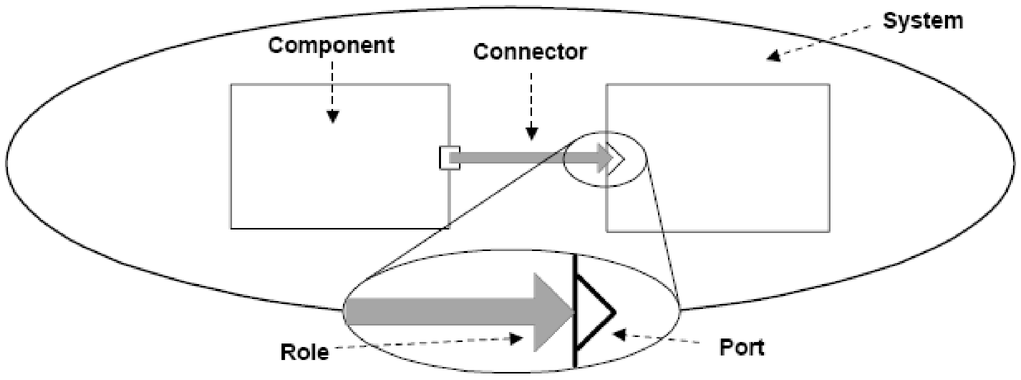


Figure 4-1 : Entities of ACME

Components represent computational elements and data stores of a system. A component may have multiple interfaces, each of which is termed a *port*. A *port* identifies a point of interaction between the component and its environment.

Connectors also have interfaces that are defined by a set of roles. Each *role* of a connector defines a participant of the interaction represented by the connector. *Systems* are defined as graphs in which the nodes represent components and the arcs represent connectors. This is done by identifying which component ports are attached to which connector roles. A *representation map* (rep-map, also called *attachment* in Figure 4-2) defines this correspondence. In the simplest case a rep-map provides an association (an *attachment*) between internal ports and external ports (or, for connectors, between internal roles and external roles).

Each of the seven entity types can be annotated with a property list. Properties document details of an architecture relevant to its design and analysis. A property has a name, an optional type, and a value. A simple example of a Client-Server system in ACME is presented in Figure 4-2.

```
System simple_cs = {
  Component client = { Port sendRequest }
  Component server = { Port receiveRequest }
  Connector rpc  = { Roles {caller, callee} }
  Attachments : {
      client.sendRequest to rpc.caller ;
      server.receiveRequest to rpc.callee }
}
```

Figure 4-2 : Example specification

Constraints determine how an architecture design is permitted to evolve over time. ACME provides a syntax to describe such constraints. Constraints can be associated with the seven entity types.
Styles allow one to define a domain-specific or application-specific design vocabulary, together with constraints on how that vocabulary can be used. The basic building block for defining styles is a type system that can be used to encapsulate recurring structures and relationships.

4.2.1.1.2 C2

A Chiron-2 (C2) architecture [81, 82] is a hierarchical network of concurrent components linked together by connectors in accordance with a set of style rules. C2 communication rules require that all communication between C2 components be achieved via message passing. C2 focuses on architectures of highly-distributed, evolvable, and dynamic systems [98].
C2 is a component- and message-based style designed to support the particular needs of applications that have a graphical user interface aspect, with the potential for supporting other types of applications as well. The C2 style supports a paradigm in which user interface components, such as dialogs, structured graphics models, and constraint managers, can more readily be reused. A sample C2 architecture is depicted in Figure 4-3.

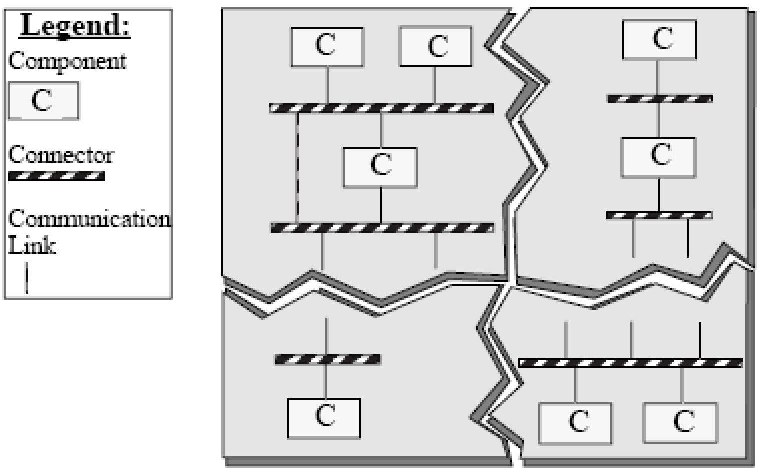
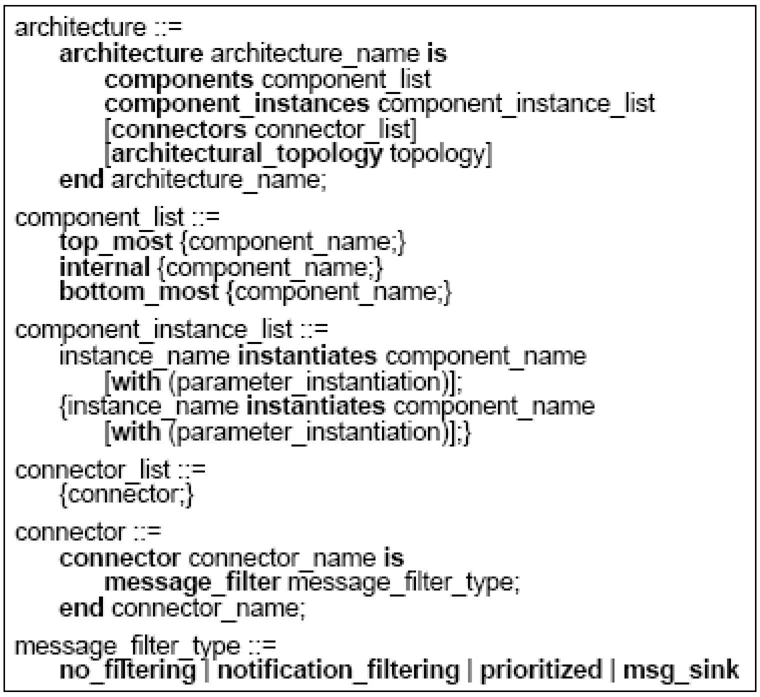


Figure 4-3 : Example C2 application

C2's ADL describes the components and the topology of the architecture. A C2 architecture in ADL is modelled as depicted in Figure 4-4 [82]:



```
architecture ::=
     architecture architecture_name is
          components component_list
          component_instances component_instance_list
          [connectors connector_list]
          [architectural_topology topology]
     end architecture_name;
component_list ::=
     top_most {component_name;}
     internal {component_name;}
     bottom_most {component_name;}
component_instance_list ::=
     instance_name instantiates component_name
          [with (parameter_instantiation)];
     {instance_name instantiates component_name
          [with (parameter_instantiation)];}
connector_list ::=
     {connector;}
connector ::=
     connector connector_name is
          message_filter message_filter_type;
     end connector_name;
message_filter_type ::=
     no_filtering | notification_filtering | prioritized | msg_sink
```

Figure 4-4 : ADL of C2

The ADL can also be used to describe the topology, or configuration, of the architecture. These features have not been discussed here but can be found in [82].

4.2.1.1.3 Aesop

Aesop [83, 84] is a system for developing style-specific architectural development [98] environments. Each of these environments supports:

- A palette of design element types (i.e., style-specific components and connectors) corresponding to the vocabulary of the style;
- Checks that compositions of design elements satisfy the topological constraints of the style;
- Optional semantic specifications of the elements;
- An interface that allows external tools to analyse and manipulate architectural descriptions;
- Multiple style specific visualisations of architectural information together with a graphical editor for manipulating them.

Aesop combines a description of a style (or set of styles) with a shared toolkit of common facilities to produce an environment called a Fable, specialized to that style (or styles) [84]. The Aesop ADL is not being actively developed at present; emphasis is shifting to ACME instead.

An Aesop architectural representation contains seven entities: components, connectors, configurations, ports, roles, representations, and bindings. Most entities correspond to an entity that is used by ACME. Each of the seven entities is represented as a C++ class. Pipeline, real-time and event-based styles can be represented using Aesop.

4.2.1.1.4 Darwin

Darwin [85, 86] is a language for describing component-based architectures. It supports a hierarchical model and is accompanied by a corresponding graphical

notation. The main contributions of Darwin are its simple yet elegant grammar, its solid concept of components and the introduction of dynamism in the specification of software architectures. Darwin focuses on architectures of highly-distributed systems whose dynamism is guided by strict formal underpinnings [98].
In Darwin, components are strongly typed first-class language primitives, supporting single inheritance. A component interface specifies what the component can provide to others, and what it requires. These ‘provide and require’ statements serve as implicit connectors; there is no explicit connector language construct. The component abstraction is deemed powerful enough to encompass connectors, i.e. if a specific type of connector is required, it can be specified as a component, with other components connected to it. Darwin supports a bind statement which is used to tie together components using their ‘provide and require’ statements. The Darwin compiler checks that connections are only made between compatible communication objects.

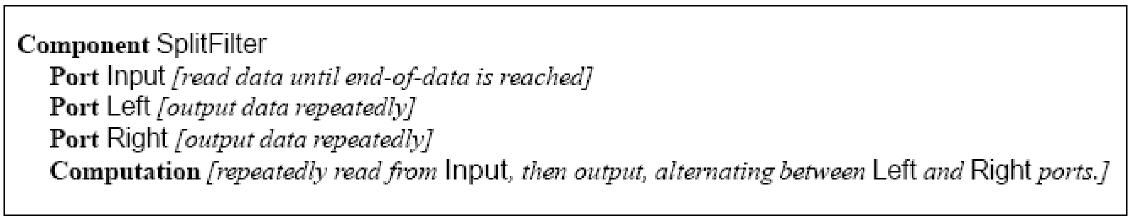
4.2.1.1.5 Rapide

Rapide [87, 88] has been designed to support component-based development of large, multi-language systems by utilising architecture definitions as the development framework. Rapide focuses on modelling and simulation of the dynamic behaviour described by an architecture [98].
Rapide, in fact, can be considered as a language framework consisting of four elements:

1. A *type language* used to define component interfaces. The language is based on a single general interface type construct together with inheritance derivations for building new interfaces from existing ones.
2. An *executable architecture definition language* which provides features for composing systems from component interfaces by defining their synchronisation and communication interconnections in terms of patterns of events.
3. A *constraint specification language* which provides constructs for abstract specification of the behaviour of a distributed system, including timing requirements.
4. A *concurrent reactive programming language* which uses types, objects, and expressions of the type language, and provides module and control structures. Its principal constructs are independent (or concurrent) reactive processes that activate when patterns of events occur during execution. These pattern-triggered processes are used to define architecture connections between components and construct component behaviours via rule-based, reactive programming.

4.2.1.1.6 Wright

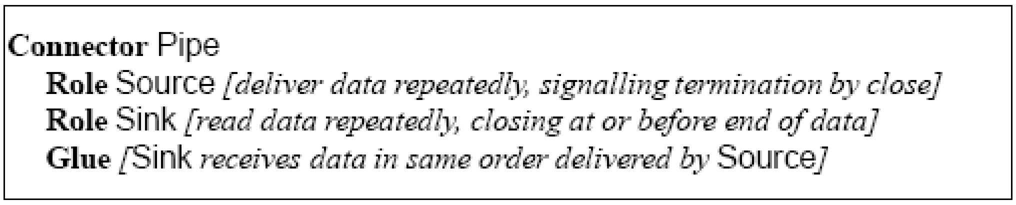
As an ADL, Wright [89] is built around the basic architectural abstractions of components, connectors, and configurations. It provides explicit notations for each of these elements, formalising the general notions of component as computation and connector as pattern of interaction. Its main focus is on modelling and analysis of the dynamic behaviour of concurrent systems [98].
The description of a component in Wright has two important parts: the interface and the computation. An interface consists of a number of ports. Each port represents an interaction in which the component may participate. The structure of a Wright component is described in Figure 4-5.



Component SplitFilter
  Port Input *[read data until end-of-data is reached]*
  Port Left *[output data repeatedly]*
  Port Right *[output data repeatedly]*
  Computation *[repeatedly read from* Input*, then output, alternating between* Left *and* Right *ports.]*

Figure 4-5 : Structure of Wright component

The structure of a Wright connector is depicted in Figure 4-6. The Glue of a connector describes how the participants work together to create an interaction.



Connector Pipe
  Role Source *[deliver data repeatedly, signalling termination by close]*
  Role Sink *[read data repeatedly, closing at or before end of data]*
  Glue *[*Sink *receives data in same order delivered by* Source*]*

Figure 4-6 : Structure of connector

In order to describe a complete system architecture, the components and connectors of a description must be combined into a configuration. A configuration is a collection of component instances combined via connectors.

4.2.1.2 Unified Modelling Language

The possibility of using the Unified Modelling Language (UML) to model the software architecture has been investigated in [90]. The authors consider two possible ways for the purpose: use UML "as is" or constrain the UML meta-model using UML's built-in extension mechanisms, like the Object Constraint Language (OCL) and stereotypes.
For using UML "as is", an architecture that is modelled in the ADL of C2 is used. To a large extent, the C2-style architecture can be successfully modelled with UML. Part of the success can be attributed to the fact that, as anticipated, many architectural concepts are found in UML. It must be noted, however, that the modelling capabilities provided by UML "as is" do not fully satisfy the structural needs of architectural description for two key reasons. First, UML does not provide specialised constructs for modelling architectural artefacts. For example, connectors and components must be modelled in UML "as is" using the same mechanism. Second, the rules of a given architectural style can not be modelled with UML.
The second approach, i.e., constraining the UML meta-model, can also be employed to constrain the UML to enforce the rules of the C2 style in a fairly straightforward fashion. This is because many C2 concepts are found in UML (cf. the example architecture depicted in Figure 4-7).

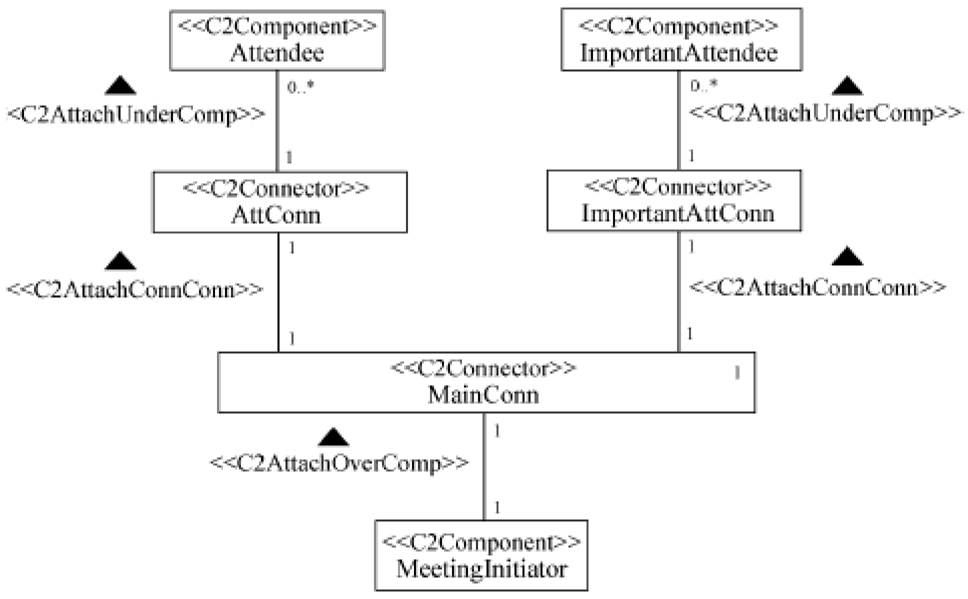


Figure 4-7 : Example architecture

This architecture modelling strategy has also drawbacks. It is heavily reliant on OCL, whose formality may hinder wide adoption of the strategy even though end users of the constrained UML model typically will not need to write OCL constraints. OCL is a part of the standard UML definition, and it is expected that standardised UML tools will be able to process it. However, OCL is considered an uninterpreted part of UML, and UML tools may not support it to the extent needed for creating, manipulating, analysing, and evolving architectural models.

4.2.2 Architectural Design Process Approaches

In this section a meta-model that is an abstraction of various architecture design approaches is provided. This meta-model is used to analyse and compare architecture design process approaches. The meta-model is presented in Figure 4-8.

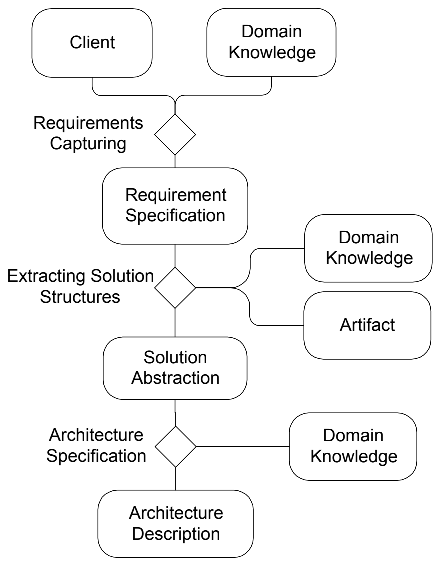


Figure 4-8 : Meta-model for architecture design process approaches

The rounded rectangles represent the concepts and the lines represent the association between these concepts. The diamond symbol represents an association relation between three or four concepts. The meanings of the concepts are as follows:

- *Client* - the stakeholders who are interested in the development of a software architecture design. A stakeholder may be a customer, end-user, system developer, system maintainer, sales manager, etc.
- *Domain Knowledge* - the area of knowledge that is applied to solve a certain problem. This term is used three times but has different meanings in different approaches. The following specialisations of this concept are distinguished (cf. Figure 4-9):
    - *Problem Domain Knowledge* refers to the knowledge of the problem from a client's perspective (includes requirement specification documents, interviews with clients, prototypes delivered by clients, etc.).
    - *Business Domain Knowledge* is the knowledge of the problem from a business process perspective (includes knowledge on the business processes and also customer surveys and market analysis reports).
    - *Solution Domain Knowledge* is the knowledge that provides the domain concepts for solving the problem. It is separate from specific requirements and the knowledge on how to produce software systems from this solution domain. This kind of domain knowledge is included in, for example, textbooks, scientific journals and manuals.
    - *General Knowledge* is the general background and experiences of the software engineer and also may include general rules of thumb.
    - *System/Product Knowledge* is the knowledge about a system, a family of systems or a product.
- *Requirement Specification* - the specification that describes the requirements for the architecture to be developed.
- *Artefact* - the artefact descriptions of a certain method. This is, for example, the description of the artefact Class, Operation, Attribute, etc. In general each artefact has a related set of heuristics for identifying the corresponding instances.
- *Solution Abstraction* - the conceptual representation of a (sub)-structure of the architecture.
- *Architecture Description* - a specification of the software architecture.

In Figure 4-8 there is a ternary association relation between the concepts *Client*, *Domain Knowledge* and *Requirement Specification*. This association means that for defining a requirement specification both client and the domain knowledge are utilised. The order of processing is not defined by this association and may differ per architecture design approach.

There is a quaternary association relation between the concepts *Requirement Specification, Domain Knowledge, Artefact* and *Solution Abstraction* which describes the structural relations between these concepts to derive a suitable solution abstraction. The ternary association relation between the concepts *Solution Abstraction*, *Architecture Description* and *Domain Knowledge* is referred to as *Architecture Specification* as it represents the specification of the architecture utilising the three concepts.

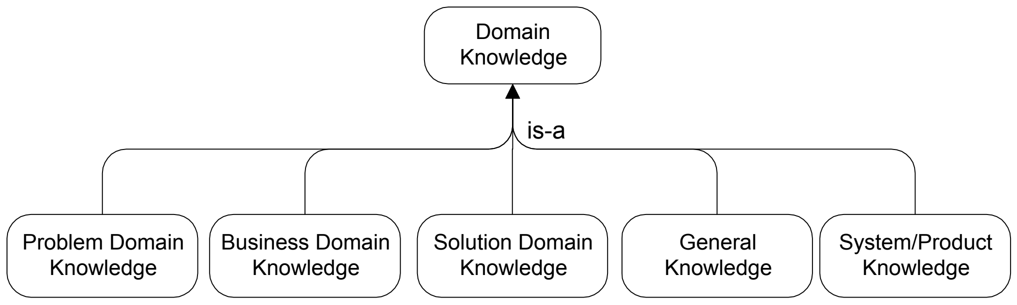


Figure 4-9 : Different specializations of the concept Domain Knowledge

Various architectural process approaches can be described as instantiations of the meta-model in Figure 4-8. Each approach will differ in the ordering of the processes and the particular content of the concepts.

A number of approaches have been introduced to identify the architectural design abstractions. We classify these approaches as *requirements-driven* and *domain-driven* architecture design approaches. The criterion for this classification is based on the adopted basis for the identification of the key abstractions of architectures. Below each approach is explained as a realisation of the meta-model from Figure 4-8.

4.2.2.1 Requirements-driven Architecture Design

The approaches in this category use *requirements,* such as *use cases*, as the primary artefacts for deriving the architectural abstractions. As discussed in section 3.3.4.1, a *use case* is defined as a sequence of actions that the system provides for *actors* [100]. Actors represent external roles with which the system must interact. The actors and the use cases together form the use case model. The use case model is meant as a model of the system's intended functions and its environment, and serves as a contract between the customer and the developers. The Unified Process [100] applies a use case driven architecture design approach. The conceptual model for the use case driven architecture design approach in the Unified Process is given in Figure 4-10. Hereby, the dashed rounded rectangles represent the concepts of Figure 4-8. For example, the concepts *Informal Specification* and the *Use Case Model* together form the concept *Requirement Specification* in Figure 4-8.

The Unified Process consists of *core workflows* that define the static content of the process and describe the process in terms of activities, workers and artefacts. The organisation of the process over time is defined by phases. The Unified Process is composed of six core workflows: *Business Modelling*, *Requirements*, *Analysis*, *Design*, *Implementation* and *Test*. These core workflows result respectively in the following separate models: *business & domain model*, *use case model*, *analysis model*, *design model*, *implementation model* and *test model*.

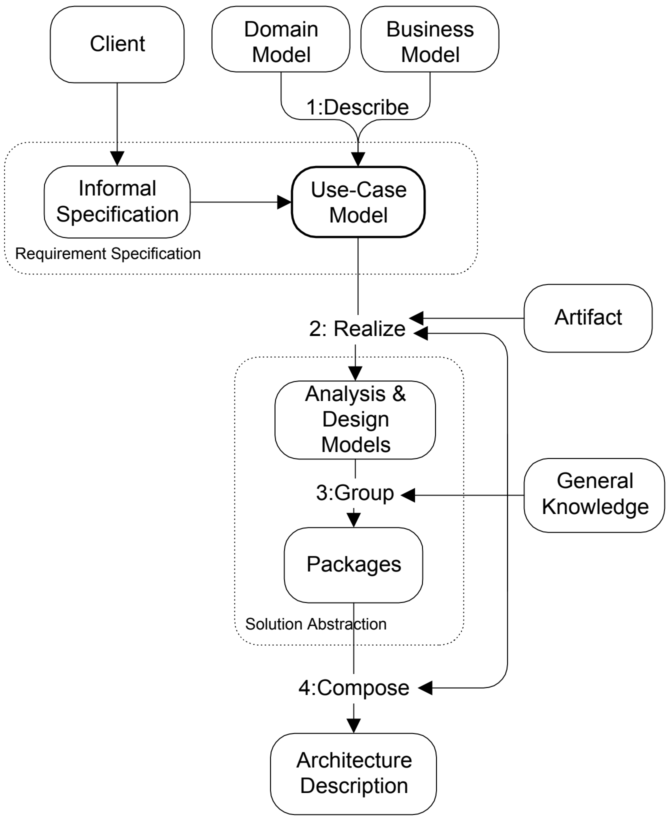


Figure 4-10 : Conceptual model of use case driven architectural design

In the requirements workflow, the client's requirements are captured as use cases which results in the use case model. This process is defined by the function *1:Describe* in Figure 4-10. Together with the informal requirement specification, the use case model forms the requirement specification. The development of the use case model is supported by the concepts *Informal Specification*, *Domain Model* and *Business Model* that are required to set the system's context. The *Informal Specification* represents the textual requirement specification. The *Business Model* describes the business processes of an organisation. The *Domain Model* describes the most important classes within the context of the domain. From the use case model the architecturally significant use cases are selected and *use case realisations* are created as it is described by the function *2:Realize*. Use case realisations determine how the system internally performs the tasks in terms of collaborating objects and as such help to identify the artefacts such as classes. The use case realisations are supported by the knowledge of the corresponding artefacts and the general knowledge. This is represented by the arrows directed from the concepts *Artefact* and *General Knowledge* respectively, to the function *2:Realize*. The output of this function is the concept *Analysis & Design Models*, which represents the identified artefacts after use case realisations.

The analysis and design models are then grouped into *packages* which is represented by the function *3:Group*. The function *4:Compose* represents the definition of interfaces between these packages resulting in the concept *Architecture Description*. Both functions are supported by the concept *General Knowledge*.

4.2.2.2 Domain-driven Architecture Design

Domain-driven architecture design approaches derive the architectural design abstractions from domain models. The conceptual model for this domain-driven approach is presented in Figure 4-11.

Domain models are developed through a domain analysis phase represented by the function *2:Domain Analysis*. Domain analysis can be defined as the process of identifying, capturing and organising domain knowledge about the problem domain with the purpose of making it reusable when creating new systems [101]. The function *2:Domain Analysis* takes as input the concepts *Requirement Specification* and *Domain Knowledge* and results in the concept *Domain Model.* Note that both the concepts *Solution Domain Knowledge* and *Domain Model* in Figure 4-11 represent the concept *Domain Knowledge* in the meta-model of Figure 4-8.

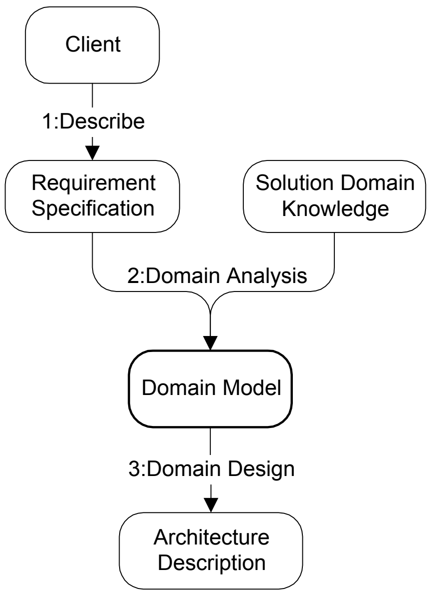


Figure 4-11 : Conceptual model for Domain-Driven Architecture Design

The domain model may be represented using different representation forms such as classes, entity-relation diagrams, frames, semantics networks, and rules. Several *domain analysis* methods have been published, e.g., [102], [103], [101], [104] and [105]. Two surveys of various domain analysis methods can be found in [106] and [107]. In [105] a more recent and extensive up-to-date overview of domain engineering methods is provided.

In this section we are mainly interested in the approaches that use the domain model to derive architectural abstractions. In Figure 4-11, this is represented by the function *3:Domain Design.* In the following we consider two domain-driven approaches, namely product-line architecture design and pattern-driven architecture design, that derive the architectural design abstractions from domain models.

4.2.2.2.1 Product-line Architecture Design

In the product-line architecture design approach, an architecture is developed for a *software product-line* that is defined as a group of software-intensive products sharing a common, managed set of features that satisfy the needs of a selected market or mission area [78]. A *software product line architecture* is an abstraction of the architecture of a related set of products. The product-line architecture design approach focuses primarily on the reuse within an organisation and involves *core asset development* and *product development*. The core asset base often includes the architecture, reusable software components, requirements, documentation and specification, performance models, schedules, budgets, and test plans and cases [108], [109], [78]. The core asset base is used to generate or integrate products from a product line.

The conceptual model for product-line architecture design is shown in Figure 4-12. The function *1:Domain Engineering* represents the core asset base development. The function *2:Application Engineering* represents the product development from the core asset base.

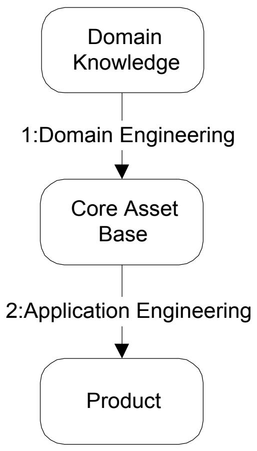


Figure 4-12 : A conceptual model for a Product-Line Architecture Design

Note that various software architecture design approaches can be applied to provide a product-line architecture design. In the following section we describe the DSSA approach that follows the conceptual model for product-line architecture design in Figure 4-12.

4.2.2.2.1.1 Domain Specific Software Architecture Design

The *domain-specific software architecture* (DSSA) [110] [111] may be considered as a multi-system scope architecture, that is, it derives an architectural description for a family of systems rather than a single-system. The conceptual model of this approach is presented in Figure 4-13. The basic artefacts of a DSSA approach are the *domain model*, *reference requirements* and the *reference architecture*. The DSSA approach starts with a domain analysis phase on a set of applications with common problems or functions. The analysis is based on *scenarios* from which functional requirements, data flow and control flow information is derived. The *domain model* includes scenarios, domain dictionary, context (block) diagrams, ER diagrams, data flow models, state transition diagrams and object models.

In addition to the domain model, *reference requirements* are defined that include functional requirements, non-functional requirements, design requirements and implementation requirements and focus on the solution space. The domain model and the reference requirements are used to derive the *reference architecture*. The DSSA process makes an explicit distinction between a *reference architecture* and an *application architecture*. A reference architecture is defined as the architecture for a family of application systems, whereas an application architecture is defined as the architecture for a single system. The application architecture is instantiated or refined from the reference architecture. The process of instantiating/refining and/or extending a reference architecture is called *application engineering*.

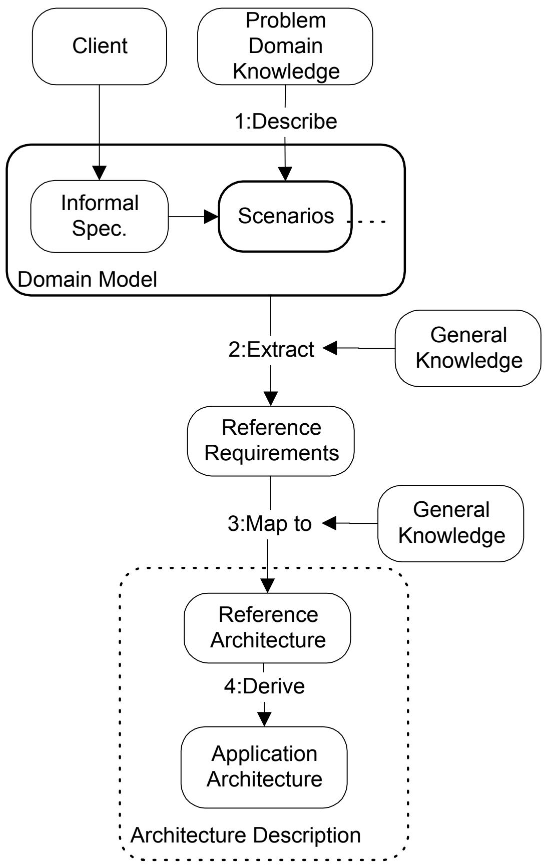


Figure 4-13 : Conceptual model for Domain Specific Software Architecture (DSSA) approach

4.2.2.2.2 Pattern-driven Architecture Design

Christopher Alexander's idea on pattern languages for systematically designing buildings and communities in architecture [112] has been adopted by the software community and led to the so-called software *design patterns* [113]. Similar to the patterns of Alexander, software design patterns aim to codify and make reusable a set of principles for designing quality software. The software design patterns are applied for the design phase, though, the software community has started to define and apply patterns for the other phases of the software development process. At the implementation phase patterns or idioms [114] have been defined to map object-oriented designs to object-oriented language constructs. Others have defined patterns for the analysis phase in which patterns are applied to derive analysis models [115]. Patterns have also been applied at the architectural analysis phase of the software development process [116, 117]. Architectural patterns are similar to the design patterns but focus on the gross-level structure of the system and its interactions. Sometimes architectural patterns are also called *architectural styles* [77, 118]. An architectural pattern is not the architecture itself, as it is often mistaken, but rather it is just an abstract representation at the architectural level [76, 119].

Pattern-driven architecture design approaches derive the architectural abstractions from patterns. Figure 4-14 depicts the conceptual model for this approach.

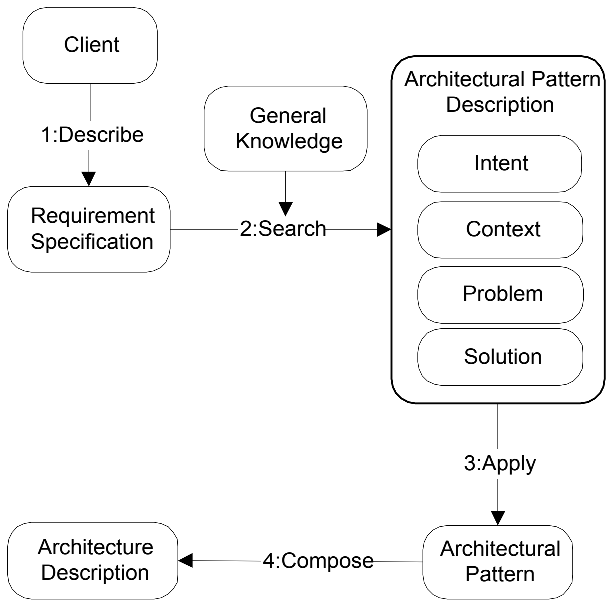


Figure 4-14 : Conceptual Model for a Pattern-Driven Architecture Design

The concept *Requirement Specification* represents a specification of a problem that may be solved using a pattern. The function *Search* represents the process for searching a suitable pattern for the given problem description and is supported by the concept *General Knowledge*.

The concept *Architectural Pattern Description* represents a description of an architectural pattern. It consists mainly of four sub-concepts[62]: *Intent*, *Context*, *Problem*, and *Solution*. The concept *Intent* represents the rationale for applying the pattern. The concept *Context* represents the situation that gives rise to the problem. The concept *Problem* represents the recurring problem arising in the context. The concept *Solution* represents a solution to the problem in the form of an abstract description of the elements and their relations. For the identification of the pattern the intent of the available patterns is scanned. If the intent of a pattern is found relevant for the given problem then the context description (*Context*) is analysed. If this also matches the context of the given problem, then the process follows with the function *3:Apply*. Thereby the sub-concept *Solution* is utilised to provide a solution to the problem. The concept *Architectural Pattern* represents the result of the function *3:Apply*. Finally, the function *4:Compose* represents the incorporation of the architecture pattern into the architecture description.

4.2.3 Architecture Evaluation Methods

As discussed earlier, software architecture forms one of the key artefacts in the entire software development life cycle since it embodies the earliest design decisions and includes the gross-level components that directly impact the subsequent analysis, design and implementation. Accordingly, it is important that the architecture design supports the software system qualities required by the various stakeholders. For ensuring the quality factors the common assumption is that identifying the fundamental concerns for architecture design is necessary and various architecture design methods have been introduced for this purpose. To verify that the right concerns have been

[62] There are other sub-concepts but we consider these four sub-concepts as important for the identification of the architectural abstractions.

identified generally static analysis of formal architectural models is conducted or a set of architecture analysis methods are adopted. Next we discuss some of these architecture evaluation methods.

4.2.3.1 Software Architecture Analysis Method (SAAM)

In [91] a comprehensive survey is given of the various software architecture design analysis methods that have been proposed so far. Among these methods the Software Architecture Analysis Method (SAAM) can be considered as a mature method which has been validated in various cases studies. Other methods such as SAAMCS, ESAAMI, SAAMER and ATAM are based on or adopt the concepts used in this method [91]. The basic activities of SAAM are illustrated in Figure 4-15.

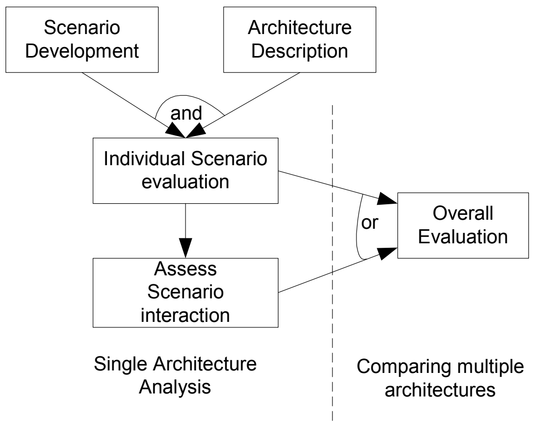

Scenario Development
Architecture Description
and
Individual Scenario evaluation
Assess Scenario interaction
or
Overall Evaluation
Single Architecture Analysis
Comparing multiple architectures

Figure 4-15 : SAAM inputs and activities [120]

SAAM takes as input a problem description, requirements statement and architecture descriptions. The steps of SAAM are as follows [120] :

1. *Describe candidate architecture:* The candidate architecture is described which includes the system's computation and data components, as well as all component relationships, sometimes called connectors.
2. *Develop scenarios:* Development of scenarios for various stakeholders; the scenarios illustrate the kinds of activities the system must support and the anticipated changes that will be made to the system over time.
3. *Perform scenario evaluations:* Scenarios are categorised into direct and indirect scenarios. For each indirect task scenario the required changes to the architecture are listed and the cost of performing these changes is estimated. A modification to the architecture means that either a new component or connection is introduced or an existing component or connection requires a change in its specification.
4. *Reveal scenario interaction:* Different indirect scenarios that require changes to the same components or connections are said to interact with respect to the corresponding component. Determining scenario interaction is a process of identifying scenarios that affect a common set of components. Scenario interaction measures the extent to which the architecture supports an appropriate separation of concerns. Semantically close scenarios should interact at the same component. Semantically distinct scenarios that interact indicate an improper decomposition.
5. *Overall evaluation:* Finally, each scenario and the scenario interactions are weighted in terms of their relative importance and this weighting used to determine an overall ranking. The weighting chosen reflects the relative importance of the quality factors that the scenarios manifest.

4.2.3.2 The Architecture Trade-Off Analysis Method (ATAM)

The Architecture Trade-Off Analysis Method (ATAM), as presented in [92], is based on SAAM, with an explicit quality model. ATAM uses a utility tree and, depending on the qualities that are important, the appropriate attribute characteristics are used. The steps that are used in ATAM to analyse the software architecture are as follows.

Presentation:

1. *Present the ATAM:* The evaluation leader describes the evaluation method to the assembled participants, tries to set their expectations, and answers questions they may have.
2. *Present business drivers:* A project spokesperson (ideally the project manager or system customer) describes what business goals are motivating the development effort and hence what will be the primary architectural drivers (e.g., high availability or time to market or high security).
3. *Present architecture:* The architect will describe the architecture, focusing on how it addresses the business drivers.

Investigation and Analysis:

4. *Identify architectural approaches:* Architectural approaches are identified by the architect, but are not analysed.
5. *Generate quality attribute utility tree:* The quality factors that comprise system "utility" (performance, availability, security, modifiability, usability, etc.) are elicited, specified down to the level of scenarios, annotated with stimuli and responses, and prioritised.
6. *Analyse architectural approaches:* Based upon the high-priority factors identified in Step 5, the architectural approaches that address those factors are elicited and analysed (for example, an architectural approach aimed at performance). During this step architectural risks, sensitivity points, and trade-off points are identified (cf. Figure 4-16).

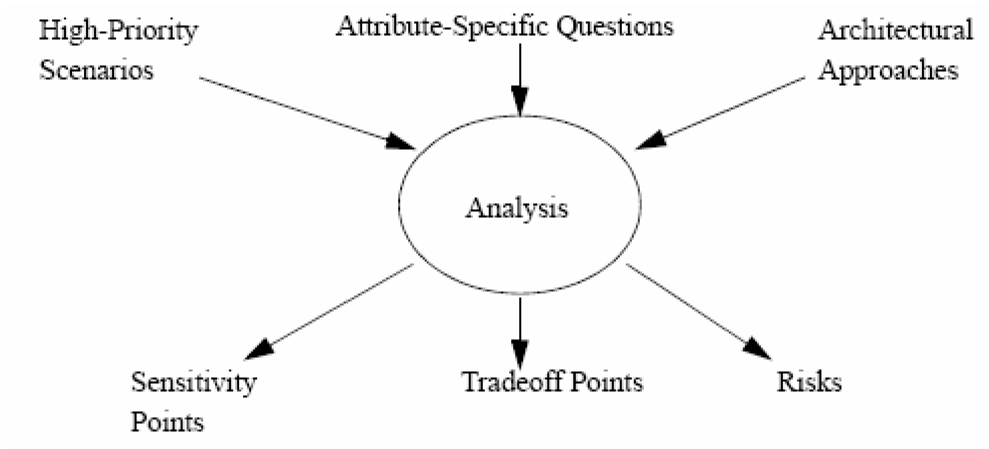


Figure 4-16 : Concepts interaction in ATAM

7. *Brainstorm and Prioritise Scenarios:* Once the scenarios have been collected, they must be prioritised. This is typically done via a voting procedure where each stakeholder is allocated a number of votes equal to 30% of the number of scenarios, rounded up.

8. *Analyse Architectural Approaches:* In this step, step 6 is reiterated. Mapping the highest ranked newly generated scenarios onto the architectural artefacts thus far uncovered. Assuming Step 7 didn't produce any high-priority scenarios that were not already covered by previous analysis, Step 8 is a testing activity: This step is expected to be uncovering little new information.
9. *Present Results:* Finally, the collected information from the ATAM needs to be summarised and presented back to the stakeholders.

4.2.3.3 SAAM Founded on Complex Scenarios (SAAMCS)

SAAMCS [93] is alternative method for software architecture analysis. The main differences between this method and SAAM are in the way in which we arrive at the scenarios and in the way in which we evaluate their impact. The process steps of SAAMCS are depicted in Figure 4-17. The first two steps: architecture description and scenario development are performed in parallel. The goal of the latter is to identify complex scenarios. A measurement instrument is used to measure the complexity of the scenarios. The goal of the measurement instrument is to provide insight into the complexity of scenarios for administrative systems. A distinction is made between the effect of a scenario on the system and on the environment. A couple of factors are defined to measure the impact on the system and on the environment. The results of this measurement instrument are used to evaluate the scenarios in the third step.

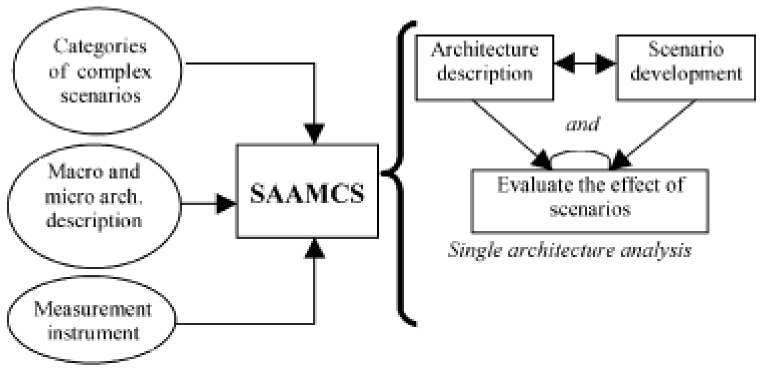
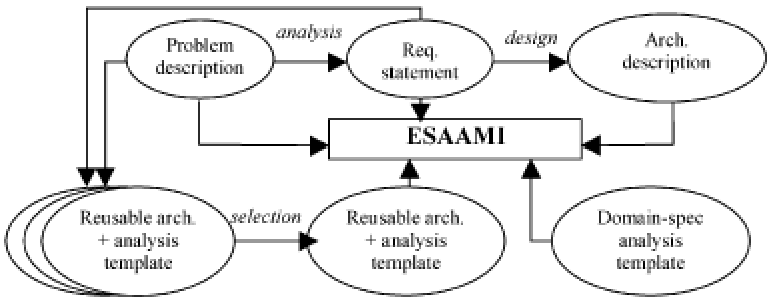

Categories of complex scenarios
Macro and micro arch. description
Measurement instrument
SAAMCS
Architecture description
Scenario development
and
Evaluate the effect of scenarios
Single architecture analysis

Figure 4-17 : Inputs and activities of SAAMCS [91]

4.2.3.4 Extending SAAM by Integration in the Domain (ESAAMI)

ESAAMI, the integration of SAAM in domain-centric and reuse-based development processes, is presented in [94]. The inputs to the ESAAMI process are depicted in Figure 4-18.



Problem description
analysis
Req. statement
design
Arch. description
ESAAMI
Reusable arch. + analysis template
selection
Reusable arch. + analysis template
Domain-spec analysis template

Figure 4-18 : Inputs of ESAAMI [91]

The SAAM process had to be made reuse-aware for the execution of the architecture analysis itself and overcoming the ad-hoc way of doing reuse. ESAAMI provides proto-scenarios, generic descriptions of reuse situations or interactions with the system. A proto-scenario may be classified as direct or indirect like a conventional SAAM scenario. The evaluation of the scenarios can be facilitated by providing protocols of earlier analyses in different projects as well as proto-evaluations. A proto-evaluation describes, by example, how the scenario can be performed using a set of abstract architecture elements. Hints are associated with each scenario indicating which architectural structures would make the scenario convenient to handle. Similar to the evaluation of the scenarios themselves, the analysis of scenario interactions can also benefit from protocols of earlier analyses in the same application domain. These elements are combined to form an analysis template.

The reuse of a SAAM analysis template in a domain-centric development process allows exploitation of the knowledge and experience from the specific application domain to increase the relevance of the analysis results.

Unlike domain-specific proto-scenarios, the proto-scenarios in an architecture-specific analysis template may refer to characteristics and elements of the scrutinised architecture. However, similar to the situation described above, they are still generic with respect to the considered application. The deployment of an architecture-specific analysis template supports a focused evaluation of characteristics relevant to the considered architecture, thus increasing the expressiveness of the analysis results.

4.2.3.5 SAAM Evolution and Reusability (SAAMER)

This framework [95] contains a set of architectural views that were developed to assess software architectures for evolution and reuse built upon SAAM. The framework is used to model different types of information, namely, stakeholder information, architecture information, quality information, and scenarios.

SAAMER considers the following architectural views as critical: static, map, dynamic, and resource. The static view integrates and extends SAAM to address the classification and generalisation of a system's components and functions and the connections between components. These extensions facilitate the estimation of cost or effort required for changes to be made. The dynamic view is appropriate for the evaluation of the behaviour aspect, to validate the control and communication to be handled in an expected manner. The mapping between components and functions could reveal the cohesion and coupling aspects of a system [91].

The SAAMER process consists of four steps. The first step involves gathering information about stakeholders, software architecture quality, and scenarios; modelling usable artefacts and undertaking analysis. The second step involves evaluation of the architecture while the later last steps are similar to those of SAAM.

4.2.3.6 Architecture Level Prediction of Software Maintenance (ALPSM)

This approach [96] aims to estimate the required maintenance effort for a software system during architectural design. The estimated effort can be used to compare two architecture alternatives or to balance maintainability against other quality attributes.
The approach takes as input the requirement specification, the design of the architecture, expertise from software engineers and historical maintenance data. The main output of the approach is an estimation of the required maintenance effort of the system built based on the software architecture. The maintenance profile is a second output from the method. The profile contains a set of scenario categories and a set of scenarios for each category with associated weighting and analysis (scripting) results.
ALPSM process consists of six steps:

1. *Identify categories of maintenance tasks:* The categories are defined based on the application or domain description.
2. *Synthesise scenarios:* For each of the maintenance categories, a representative set of concrete scenarios is defined.
3. *Assign each scenario a weight:* The prediction method requires probability estimates, i.e. weights, for each scenario. These probabilities are used for balancing the impact on the prediction of more occurring and less occurring maintenance tasks.
4. *Estimate the size of all elements:* To estimate the maintenance effort, the size of the architecture needs to be known and the sizes of the affected components need to be known.
5. *Script the scenarios:* The maintainability of the architecture is estimated by scripting the scenarios.
6. *Calculate the predicted maintenance effort:* The value is a weighted average for the effort for each maintenance scenario. Based on that, one can calculate an average effort per maintenance task.

4.3 Aspect-Oriented Approaches

Like the non-AO approaches, we categorise the aspect-oriented architecture design approaches into:

- Architectural modelling approaches represented by the Perspectival Concern-Space Framework [121] and DAOP-ADL [122];
- Architectural process approaches represented by Aspect-Oriented Generative Approaches [123, 124] and TranSAT [125];
- Architecture Evaluation approaches represented by Aspectual Software Architecture Analysis Method [126].

Each approach in each of the three categories is described in terms of its method for architecture design, the artefacts and the process, if applicable.

4.3.1 Architectural Modelling Approaches

4.3.1.1 The Perspectival Concern-Space (PCS) Framework

PCS Method

The Perspectival Concern-Space (PCS) [121] is a technique for depicting concerns of multiple dimensions in an architectural view consisting of one or more models and diagrams. A perspective is a "way of looking" at a multidimensional space of software concerns from one specific viewpoint. Figure 4-19 gives an idea of the key concepts used in the PCS Framework and summarises the combination of the realisations of the conceptual frameworks for Multi Dimension Separation of Concerns (MDSOC) and IEEE-Std-1471, and UML.

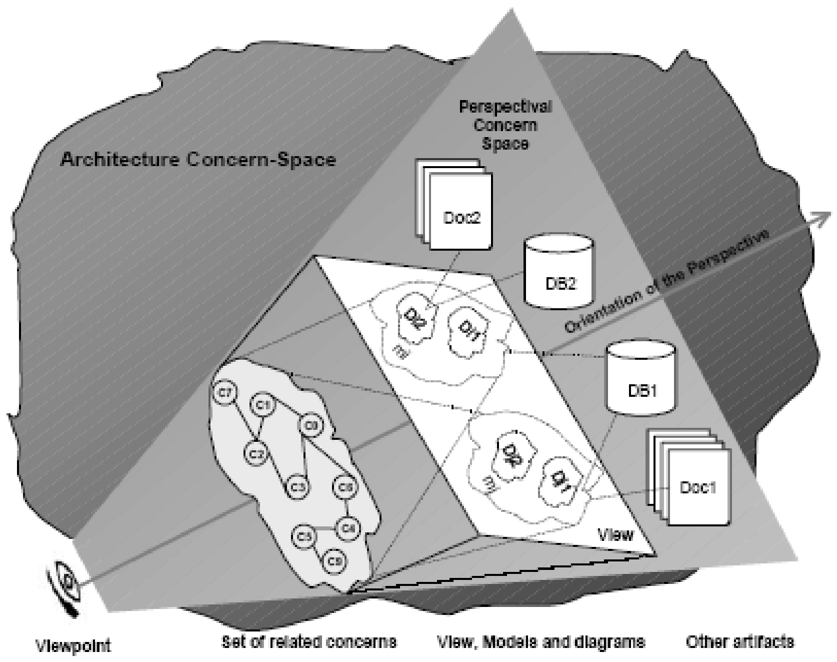

Architecture Concern-Space
Perspectival Concern Space
Doc2
DB2
Orientation of the Perspective
DB1
Doc1
View
Viewpoint
Set of related concerns
View, Models and diagrams
Other artifacts

Figure 4-19 : A Perspectival Concern-Space in Overview

PCS Artefacts

The Aspect-Oriented Construction PCS is a specific type of PCS and demonstrates how MDSOC helps deal with software complexity by supporting the composition of independent software components along different interaction concerns.

A UML Space is used for Aspect-Oriented Modelling (AOM), see Figure 4-20 for a high level view and Figure 4-21 for a low level view. The approach proposes two packages as extension to UML. The AOM Core package specifies the basic AOM constructs necessary to model aspect-oriented software. The AOM Data Types package defines basic data types. The Aspect-Oriented Model can be mapped to an AspectJ program, which is a manual process.

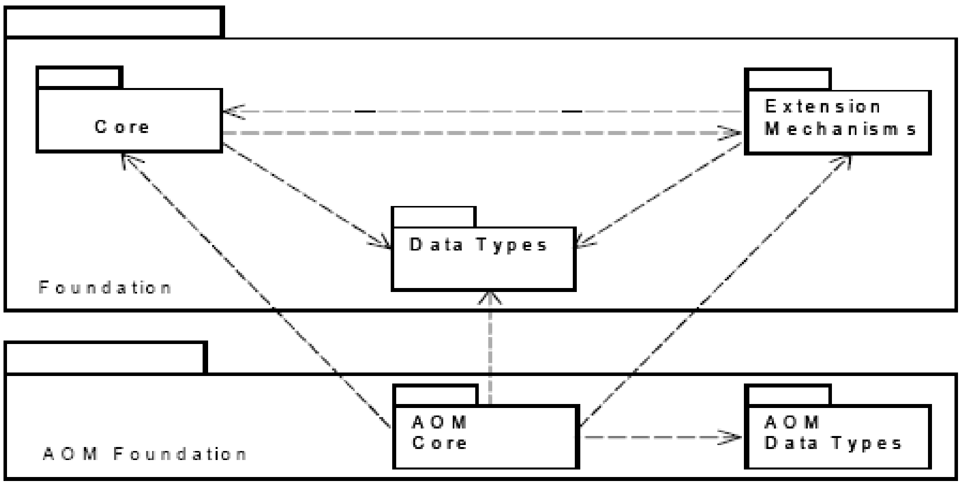


Figure 4-20 : High-Level Package View of the UML Space for AOM

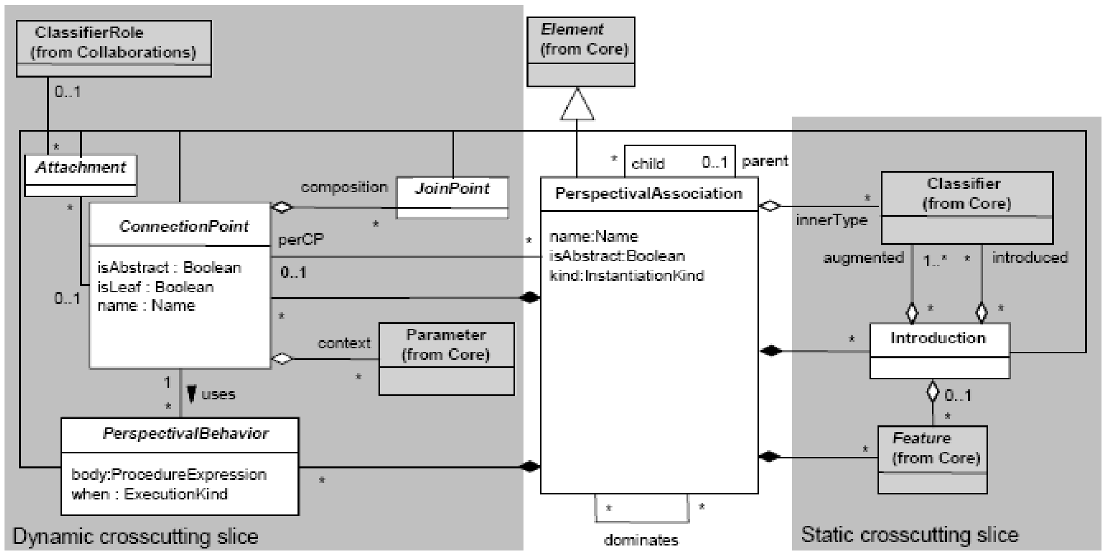


Figure 4-21 : The UML Space for AOM — A Low-Level View of AOM Core

With the tool ConcernBase it is possible to translate the UML models to Structural Architecture Description Language (SADL). SADL is a particular ADL that focuses on understanding, specifying and refining the representation of structural concerns in complex software systems. SADL is different from other ADLs, such as Wright, in that it supports structural decomposition at multiple levels. This is called refinement of high-level system structures in the SADL terminology.

4.3.1.2 DAOP-ADL

DAOP-ADL Method

DAOP-ADL [122] is an XML-based architecture description language to describe the architecture of an application in terms of a set of *components*, a set of *aspects* and the interconnections among them. As shown in Figure 4-22, these interconnections are structured in two different kinds of *composition constraints*:

1. The *componentCompositionRules* describe the rules that drive the composition of components
2. The *aspectEvaluationRules*, which are the equivalent to aspect pointcuts in aspect-oriented programming languages, describe the weaving rules between components and aspects.

The dependencies among non-orthogonal concerns are expressed by means of shared *properties*. The language also includes all the information needed to deploy the application (the *deployment information*) and the application *initial context*. DAOP-ADL is part of the CAM/DAOP approach described in section 5.4.16 of this document. The goal of this approach is specification of component and aspect based distributed applications.

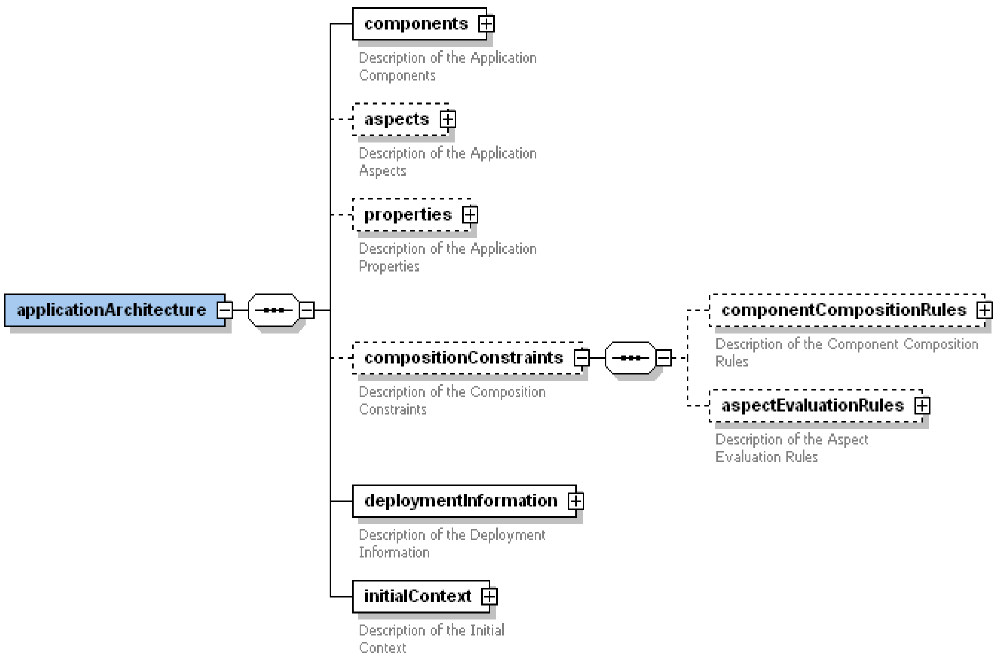

components
Description of the Application Components
aspects
Description of the Application Aspects
properties
Description of the Application Properties
applicationArchitecture
compositionConstraints
Description of the Composition Constraints
componentCompositionRules
Description of the Component Composition Rules
aspectEvaluationRules
Description of the Aspect Evaluation Rules
deploymentInformation
Description of the Deployment Information
initialContext
Description of the Initial Context

Figure 4-22 : The structure of the DAOP-ADL language

DAOP-ADL Artefacts

The artefacts in DAOP-ADL are the components and the aspects that set up an application. These artefacts are described by means of their provided and required interfaces.

```
<component role = “chat”>
  <providedInterface> ChatProv.xml </providedInterface>
  <requiredInterface> ChatReq.xml </requiredInterface>
  <implementations>
    <implementation>
      <name>chat1</name>
      <language>java</language>
      <class>Chat.class</class>
    </implementation>
  </implementations>
</component>
```

```
<providedInterface>
  <method name = “sendText”>String</method>
  <method name = “sendColour”>Colour</method>
  …
</providedInterface>
```

```
<requiredInterface>
  <message name = “sendText”>
    <targetRole>chat</targetRole>
    <parameter>String</parameter>
  <message>
  …
</requiredInterface>
```

```
<aspect role = “persistence”>
  <evaluatedInterface> PersistenceEval.xml </evaluatedInterface>
  <implementations>
    <implementation>
      <name>persistence1</name>
      <language>java</language>
      <class>Persistence.class</class>
    </implementation>
  </implementations>
</aspect>
```

```
<evaluatedInterface>
  <method name = “sendText”></method>
  …
</evaluatedInterface>
```

```
<aspect role = “authentication”>
  …
  <setProperty>username</setProperty>
</aspect>

<aspect role = “userfilter”>
  …
  <getProperty>username</getProperty>
</aspect>

<property name = “ username“>
  <type>String</type>
</property>

<compositionRules>
   <compCompositionRules>…</compCompositionRules>
   <aspectCompositionRules>
     <aspectRule>
       <targetCompRole>chat</targetCompRole>
        <BEFORE_NEW>                         //rule to apply the authentication aspect
          <aspectList>authentication</aspectList>
        </BEFORE_NEW>
     </aspectRule>
     <aspectRule>
       <sourceCompRole>chat</targetCompRole>
       <targetCompRole>chat</targetCompRole>
       <BEFORE-SEND>                         //rule to apply the persistence aspect
          <messages>sendText</messages>
          <aspectList>persistence</aspectList>
       </BEFORE_SEND>
       <BEFORE_RECEIVE>                      //rule to apply the userfilter aspect
          <messages>sendText</messages>
          <aspectList>userfilter</aspectList>
       </BEFORE_RECEIVE>
      </aspectRule>
   </aspectCompositionRules>
</compositionRules>
```

Figure 4-23 : Component and aspect XML description

The output artefact of using the DAOP-ADL language is an XML-based document (cf. Figure 4-23) that contains the description of all the components and aspects that may be instantiated in the final application and the set of plug-compatibility rules that determines how the entities in the application (core functionality and concerns) are to be composed.

DAOP-ADL Process

The process for DAOP-ADL based architecture design is as follows. First, all the components and aspects that may be instantiated in an application are described using the component and aspect description section of the DAOP-ADL language. If COTS components and aspects are going to be used the information can be automatically generated by inspecting the component and aspect binary code. Then, the information about the composition among components, the relationships among aspects and the composition between components and aspects is provided in the composition constraints section of the language. After this, this information can be validated to check if some mismatches exists.

The tool *Component and Aspect Repository* is used to register COTS components and aspects. The tool automatically generates the description of loaded components and aspects using the syntax of the DAOP-ADL language. Then, the tool *Aspect Specification and Validation* is employed. It has the DAOP-ADL language as its back-end and supports description and validation of the software architecture of the application.

DAOP-ADL is used in conjunction with CAM/DAOP. CAM is a design model used to design component and aspect based applications in UML. DAOP, on the other hand, is a component and aspect based platform that loads the architecture description provided via the DAOP-ADL specification to obtain the information needed to instantiate components and aspects and to perform the dynamic composition of components and aspects at runtime. DAOP-ADL supports traceability in the sense that all the components and aspects identified during the design of the application with CAM are described in XML using the DAOP-ADL language. Additionally, all the information provided with DAOP-ADL is directly used at runtime, closing the "gap" between the design and implementation of component and aspect-based applications.

4.3.2 Architectural Process Approaches

4.3.2.1 Aspect-Oriented Generative Approaches (AOGA)

AOGA Method

Aspect-Oriented Generative Approaches (AOGA) [123, 124] is an architecture-centric approach that was initially outlined in [127] and further extended in [123, 124, 128] with the purpose of supporting developers of multi-agent systems (MAS) with domain-specific languages, modelling notations, and code generation tools. Although the approach has been initially applied to the MAS domain, the concepts introduced are general and not limited to the MAS domain. The basic idea of the approach is to promote the integration of generative and aspect-oriented technologies in order to facilitate the domain modelling, the architectural specification and the code generation of crosscutting features starting from early development stages. Aspects can be captured and specified in preliminary development stages, even before the architectural stage. In this sense, AOGA has defined extensions to feature models [129] and a new domain-specific language (DSL) in addition to a UML-based notation to express architectural aspects.

This approach covers the following life cycle phases: Domain analysis and specification, Architecture design, and Implementation. As illustrated in Figure 4-24, the first phase encompasses the specification of crosscutting features as domain aspects. The architecture design involves the use of a UML-based notation to define the architectural aspects as part of an aspect-oriented software architecture. The implementation phase includes the use of a code generator, pre-defined frameworks, pre-defined components, and code templates.

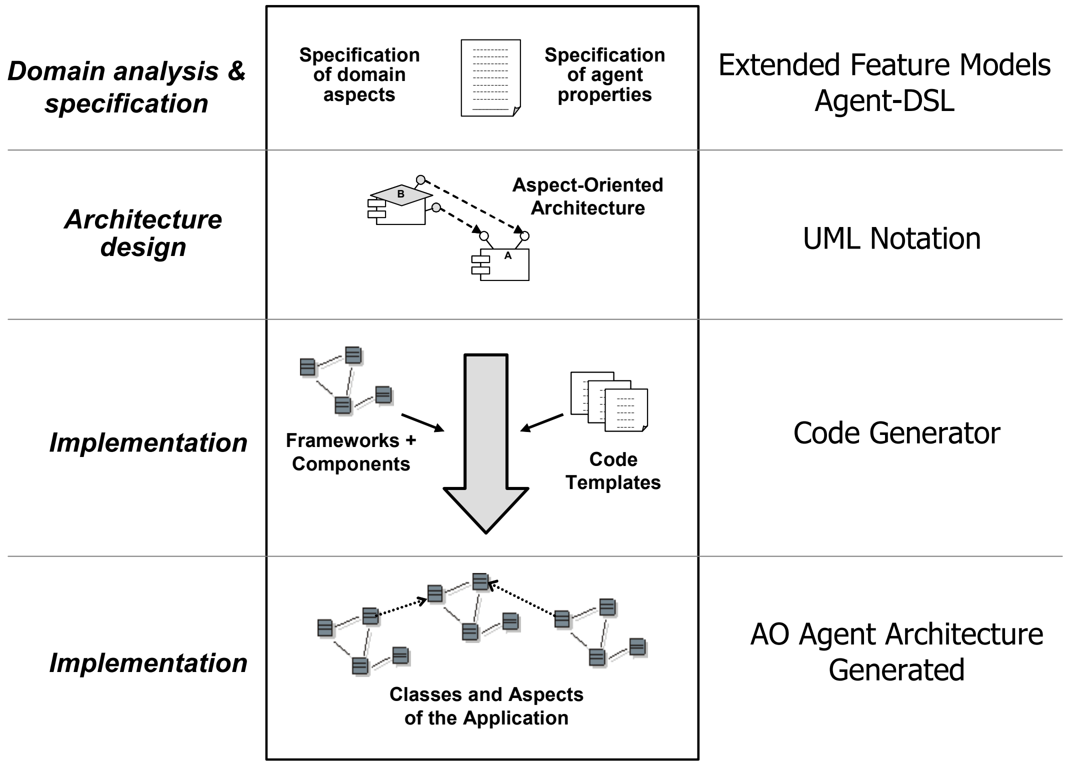

Domain analysis & specification
Specification of domain aspects
Specification of agent properties
Extended Feature Models
Agent-DSL
Architecture design
Aspect-Oriented Architecture
B
A
UML Notation
Implementation
Frameworks + Components
Code Templates
Code Generator
Implementation
Classes and Aspects of the Application
AO Agent Architecture Generated

Figure 4-24 : The development phases covered by the AOGA approach

The driving goal of the AOGA approach is to empower software developers with means to modularise crosscutting features in a stepwise fashion. This top-level goal is, in turn, decomposed into three subgoals:

- support the identification and specification of domain aspects;
- enable the identification and specification of architectural aspects;
- automate the code generation of the specified aspect-oriented architecture.

AOGA Artefacts

Extended feature models are the artefacts used in the domain analysis and specification phases. The original definition of feature models was intended to explicitly support the representation of common and variable features in a certain domain. Feature models are also used to represent different types of relationships between features. The AOGA approach extends feature models to emphasise the distinction between non-crosscutting features and crosscutting ones. Figure 4-25 presents an illustrative example of the feature models. Note that a new relation type is defined, called *crosscutting*, which makes it explicit which features a given crosscutting feature is affecting. A feature A crosscuts a feature B, when either A or one of its sub-features depends and inspects B or one of the sub-features of B. In Figure 4-25, for example, feature A is being affected by the crosscutting feature B. It is also possible to express which specific sub-features are being affected; in Figure 4-25, the features A1 and A2 are the affected sub-features. The AOGA approach also provides a domain-specific language, called Agent-DSL, which is used to collect and model both orthogonal and crosscutting features of agent-based applications. Although this is an XML-based language that is compliant with the generic AO extensions of the feature models, it is specially tailored to the MAS domain.

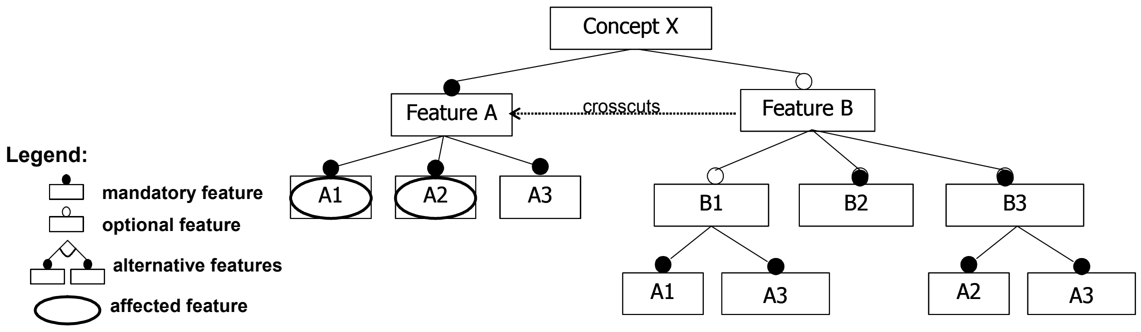


Figure 4-25 : Crosscutting feature models

For the architectural stage, AOGA has a UML-based language for specifying and communicating aspect-oriented software architectures. It provides a notation and semantics that enable architects of AO software to build models that focuses on the key components of aspect-oriented systems. The main goal is to avoid the architect to deal with design issues that are not relevant in the architectural stage. Figure 4-26 illustrates the notation elements of the architectural model that makes a distinction between *normal components* and *aspectual components*. Aspectual components (or architectural aspects) are aspects at the architecture level. Architectural aspects are UML components [130] represented as diamonds. Each of the aspectual components is related to more than one architectural component, thus representing its crosscutting nature. Note that the architectural view of an aspect suppresses all information about its inner elements.

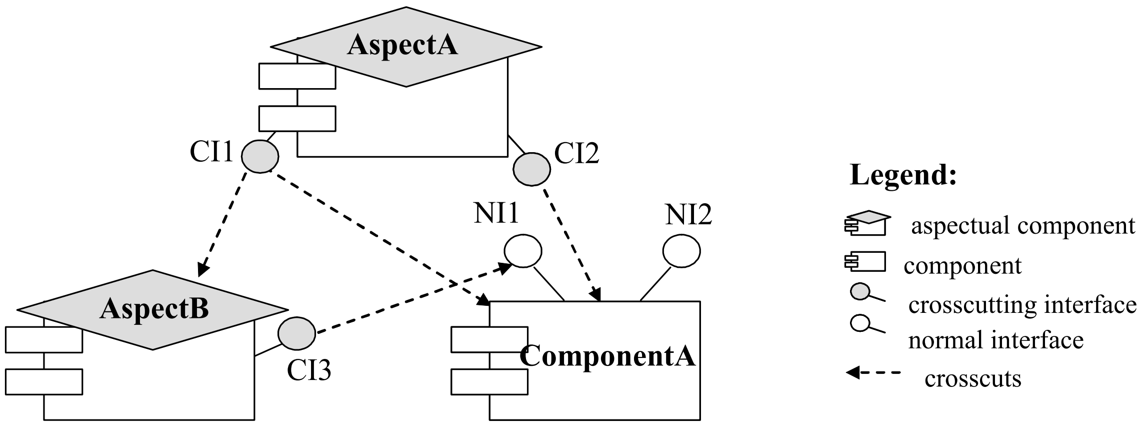


Figure 4-26 : Architectural aspects and crosscutting interfaces

Interfaces of the architectural components are also defined in a higher-level fashion. Figure 4-26 illustrates some architectural components and their interfaces. Each interface is displayed as a small circle with the interface name placed next to the circle. Each architectural component has one or more interfaces, and different components can realise the same interface. The interfaces are attached to the architectural components, and are categorised into two groups: *normal interfaces* and *crosscutting interfaces* [131]. Normal interfaces are coloured in white and crosscutting ones in grey. A crosscutting interface is different from a normal interface. The latter only provides services to other components. Crosscutting interfaces in the architectural model specify which architectural components an aspectual component affects; note that the architectural model does not declare how they are affected. An aspectual component conforms to a set of crosscutting interfaces. An aspect interface crosscuts either internal elements of architectural components or other interfaces. The first case means that the architectural aspect directly affects the internal structure or dynamic behaviour of the target component. The second case means that the aspect affects the behaviour defined by the crosscutting interface.

The output artefacts are: feature models and component models. Feature models are represented in a domain specific language called Agent-DSL. Agent-DSL uses an XML Schema to represent the features. In the architecture design phase, the aSide modelling language is used to represent the software architecture.

AOGA Process

In the AOGA process first a domain-specific language (DSL), called Agent-DSL is used to collect and model both non-crosscutting and crosscutting features of software agents. After that, the system designers specify an AO architecture for the system at hand. In this architectural phase, the designers concentrate on two main issues. First, they work on the specification of the central components of the AO system, as described above. Second, software architects define the interfaces of the architectural components in a higher-level fashion. Thus the AO architecture is centred on the definition of aspectual components to modularise the crosscutting agent features at the architectural level of abstraction. In the last step a code generator, that maps abstractions of the Agent-DSL to specific compositions of objects and aspects in the agent architecture, is used. The tool *agent architecture generator* is implemented as an Eclipse plug-in. The tool can read the agent description of the Agent-DSL. The plug-in can generate the classes that represent the elements of the XML Schema.

There is a direct trace link between a crosscutting feature in domain analysis and an aspectual component in domain design [123]. So, there is traceability support available from the domain application model to the architecture application model. In addition, an architectural aspect is mapped to a set of implementation aspects that refine that aspectual component.

4.3.2.2 TranSAT

TranSAT Method

TranSAT [125] is a framework for the specification of software evolution. TranSAT focuses on facilitating architecture evolution through realising AOSD principles in an architecture context. It proposes the incremental definition of the software architecture by weaving new architecture plan within a software architecture. Architecture specification is transformed by integrating technical concerns within the architecture.

Three challenges are addressed by TranSAT. First, the integration of a new concern should be performed by the framework. Second, a concern should be generic enough to be reusable in several contexts. Third, the integration of a new concern must not break the architecture description consistency.

TranSAT Artefacts

To solve the two first challenges, TranSAT introduces the concept of *software architecture pattern*. Though associated with a concern, it is independent of the integration context. A pattern gathers all the information needed to enable the integration of a concern and organises this information in three parts: *an architecture plan*, *a join point mask*, and *a set of transformation rules*.

A *new architecture plan* identifies a self-sufficient component assembly, which implements a given concern, and specifies its structural and behavioural properties. A new plan contains only the information related to the concern. It is defined independently from any given software architecture specification in order to enable its reuse in several contexts.

To integrate a new plan, the pattern needs plugs on the basis plan. These plugs called *join points* correspond to any locations in a component assembly to which the plan can be hooked.

Even if the pattern is independent of the integration context, it must assume some hypotheses on the software architecture that it can transform. These hypotheses are captured by the *join point mask.* It is an architecture template that determines the integration context constraints. It declares structural and behavioural preconditions that a basis plan must satisfy before the integration operation is performed. It defines the form of plug on which the new plan can be attached.

The *transformation rules* specify the operations to be carried out in order to integrate the new plan into a basis plan. These operations are applied on each of the selected join points. They specify how a concern should be integrated with a system on an architectural, structural and behavioural level.

To enable integration of concern and architecture, a weaver describes the interactions between specific software architectures and a technical concern (a pattern). The weaver binds the concern to the specific architecture. It contains the pointcut which defines where the new plan must be integrated and performs the transformation rules. It updates relevant parts of the architecture with the concern.

A (TranSAT) *pointcut* specialises the join point mask of the pattern with respect to the target basis plan. It selects, among all the join points compatible with the join point mask, a set of join points on the basis plan based on context criteria, such as architectural element names or relations.

Each technical concern that is integrated is considered to be an evolutionary step. For each evolution, a specific weaver is required to translate the join point mask to the correct set of pointcuts.

To deal with the last challenge, TranSAT introduces several levels of verification before the transformation with the join point mask, during the transformation with the addition of constraints on the transformation rules meta-model and after the transformation with tools provided by the architecture description language called SafArchie [132].

TranSAT Process

TranSAT can be viewed as a process in two distinct ways. Firstlym TranSAT supports the entire software development lifecycle as an iterative evolution process. In this view, concerns are merged together until a system is complete. This assumes that the core model is a composite concern that grows at each evolution of the architecture. Concerns are developed completely independently of the core and then integrated with the core. During iterations the core grows and evolves. Secondly, TranSAT applies to the maintenance phase of a software development lifecycle. In this case, development is finished and a core architecture and system exists. Evolution through expanding the architecture with new technical concerns could be considered to be maintenance.

Taking either view, a set of steps is outlined for using TranSAT. When abstracted away from TranSAT, these steps can be considered a high level process description for concern integration.

The integration of a new plan into a basis architecture plan is guided by the pattern's join point mask and transformation rules. The TranSAT framework determines, based on the join point mask, the set of all the possible join points, *i.e.,* the locations on the basis plan where the new plan can be attached. The architect may, however, want to impact only a few join points among this set. To select specific join points, the architect defines a pointcut expression. Similarly to AOP approaches, this expression specifies concrete architecture element names and relations in order to restrict the set of possible join points and to consequently keep only the ones that should be impacted by the transformation operations. At each chosen join point, the weaver integrates the new architecture plan according to the pattern's transformation rules. This weaving operation is a two-step process. At the first step, for each join point, the tagged elements present in both the join point mask and the transformation rules are substituted by the actual elements of the basis plan. At the second step, the weaver executes the transformation rules at each join point to finally yield the transformed software architecture plan.

In the TranSAT approach, several actors participate in the transformation process: a domain expert, an integrator, and an architect. The role of the domain expert is to define a component assembly for a given concern (*a plan*). The integrator is in charge of specifying the join point mask and the transformation rules to associate with the component assembly and form a complete software architecture pattern. The architect builds the software architecture by successively integrating patterns.

From the architect point of view, the transformation process is decomposed into four steps. First, (s)he chooses a pattern corresponding to his/her needed concern. Second, for this pattern and in accordance with the pattern's join point mask, the weaver is used to determine a set of compatible join points on the basis plan. Third, the architect defines a pointcut expression that selects only the join points to which the new plan should be attached. Fourth, the transformation rules are performed on each selected join point. The result is a new software architecture specification that contains the given concern. The architect can consider this software architecture as a new basis plan on which s/he can perform other transformations.

4.3.3 Architectural Evaluation Approaches

4.3.3.1 Aspectual Software Architecture Analysis Method (ASAAM)

ASAAM Method

The aim of ASAAM is to explicitly identify and specify architectural aspects early in the software life cycle [126]. The approach builds on scenario-based architecture analysis methods, and as such, should be considered as a complementary approach to these methods. The benefit of ASAAM is in the systematic support for the management of architectural aspects in an explicit manner.

ASAAM Artefacts

The two key artefacts in ASAAM are scenarios and architectural components. There are three types of scenarios: direct, indirect and aspectual. A direct scenario can be directly performed. Indirect scenarios require a change of a component and aspectual scenarios can either be direct or indirect and are scattered across multiple components. There are four types of components:

- Cohesive component, which is a component that is well defined and performs semantically close scenarios;
- Ill-defined component, a component consisting of several sub-components each of which performs semantically close set of scenarios;
- Tangled component, a component that performs an aspectual scenario which is either directly or indirectly performed by the component;
- Composite component, a component that includes semantically distinct scenarios but which cannot be decomposed or does not include an aspectual scenario.

The architectural aspects are the output artefacts, these aspects can be used to refactor the architecture.

ASAAM Process and Heuristics Rules

The process consists of five activities which are depicted in Figure 4-27 [126]:

1. *Candidate architecture development:* A (candidate) architecture design is provided that will be analysed with respect to the required quality factors and potential aspects.
2. *Development of scenarios:* This activity is similar to SAAM. Scenarios from various stakeholders are collected, which represent both important uses and anticipated uses of the software architecture.
3. *Individual scenario evaluation and aspect identification:* Scenarios are categorised into direct and indirect scenarios. The scenario evaluation also searches for potential architectural aspects. The application of the heuristic rules results in a further classification of the scenarios into direct scenarios, indirect scenarios, aspectual scenarios and architectural aspects. Aspectual scenarios are derived from direct or indirect scenarios and represent potential aspects.

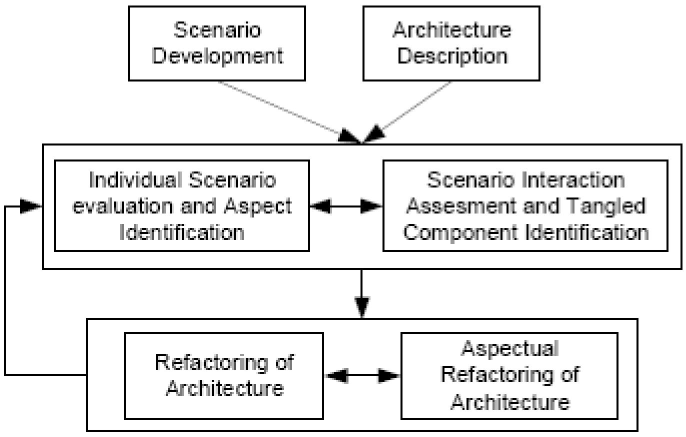
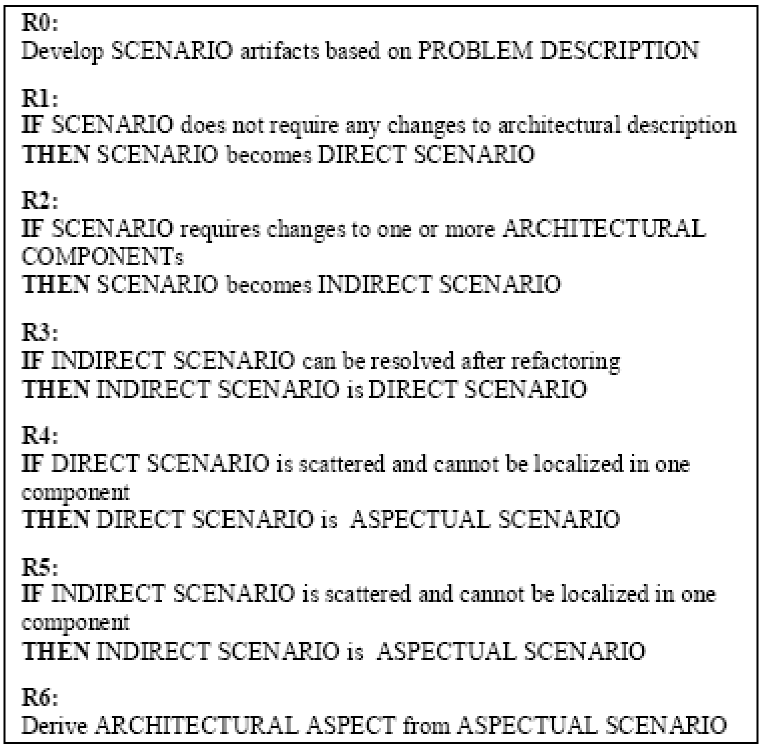

Scenario Development
Architecture Description
Individual Scenario evaluation and Aspect Identification
Scenario Interaction Assesment and Tangled Component Identification
Refactoring of Architecture
Aspectual Refactoring of Architecture

Figure 4-27 : The activities for ASAAM

4. *Scenario interaction assessment and component identification:* The goal of this activity is to assess whether the architecture supports an appropriate separation of concerns. This includes both non-crosscutting concerns and architectural aspects. For each component both direct and indirect components are analysed and categorised into cohesive component, tangled component, composite component, or ill-defined component.
5. *Refactoring the architecture:* A refactoring of the architecture is proposed based on the scenario interaction assessment and component classifications. The architectural aspects and the tangled components are explicitly described in the architecture.

ASAAM defines a set of heuristic rules to categorise scenarios into direct scenarios, indirect scenarios and architectural scenarios and aspects. This set of rules is depicted in Figure 4-28. ASAAM also defines heuristic rules to categorise the architectural components.


R0:
Develop SCENARIO artifacts based on PROBLEM DESCRIPTION

R1:
IF SCENARIO does not require any changes to architectural description
THEN SCENARIO becomes DIRECT SCENARIO

R2:
IF SCENARIO requires changes to one or more ARCHITECTURAL COMPONENTs
THEN SCENARIO becomes INDIRECT SCENARIO

R3:
IF INDIRECT SCENARIO can be resolved after refactoring
THEN INDIRECT SCENARIO is DIRECT SCENARIO

R4:
IF DIRECT SCENARIO is scattered and cannot be localized in one component
THEN DIRECT SCENARIO is ASPECTUAL SCENARIO

R5:
IF INDIRECT SCENARIO is scattered and cannot be localized in one component
THEN INDIRECT SCENARIO is ASPECTUAL SCENARIO

R6:
Derive ARCHITECTURAL ASPECT from ASPECTUAL SCENARIO

Figure 4-28 : Heuristic rules for scenario evaluation

ASAAM has been implemented as an Eclipse plug-in in the tool environment ASAAM-T [133].

4.4 Comparison

This section compares the architecture design approaches presented earlier. The comparison is performed with respect to the criteria of section 2.

4.4.1 Traceability

Preservation of traceability between the artefacts of the software lifecycle is one of the crucial qualities required for understandable and maintainable software. This criterion could be broken into two counterparts: 1) traceability of artefacts to their source of origin and change and 2) traceability between lifecycle artefacts .

Approach	**Features that Support Traceability of artefacts to their source of origin and change**	**Features that Support Traceability between lifecycle artefacts**
ACME	Not considered	ACME properties serve to document details of an architecture relevant to its design and analysis.
C2	Not considered	Architecture is refined into a partial implementation, which contains completion guidelines for developers derived from the architectural description.
Aesop	Not considered	Not considered.
Darwin	Not considered	Architectural description serves to ensure proper interconnection and communication among architectural components when they are implemented in a specific programming language.
Rapide	Not considered	Executable sublanguage that contains many common programming language control structures. Three kinds of conformance criteria are checked between system implementations and architecture: decomposition, interface conformance and communication integrity.
Wright	Not considered	Glue information in connectors is augmented with a *trace specification*, which defines a predicate that must be true for every trace of the glue, thereby restricting the set of traces permitted by the connector.
UML for Architecture	Not considered	UML tools supports the automatic generation of code and XMI-based descriptions (that can be processed by tools at runtime) from UML models
Requirements-driven AD	This document describes a conceptual model for requirements-driven AD. Support for traceability depends on specific requirements-driven AD approaches	This document describes a conceptual model for requirements-driven AD. Support for traceability depends on specific requirements-driven AD approaches
Domain-driven AD	This document describes a conceptual model for domain-driven AD. Support for traceability depends on specific domain-driven AD approaches.	This document describes a conceptual model for domain-driven AD. Support for traceability depends on specific domain-driven AD approaches.
DSSA (Product-line Driven AD)	Not considered	*Reference requirements* include functional requirements, non-functional requirements, design requirements and implementation requirements and focus on the solution space. This information is used to derive the *reference architecture*.
Pattern Driven AD	This document describes a conceptual model for pattern-driven AD. Support for traceability depends on specific pattern-driven AD approaches.	This document describes a conceptual model for pattern-driven AD. Support for traceability depends on specific pattern-driven AD approaches.
SAAM	First, the definition of scenarios for various stakeholders anticipates changes that will be made to the system over time. Then, during the evaluation of scenarios	Requirement Specification taken into account during the evaluation of the architecture.

	the required changes to the architecture are listed and the cost of performing these changes is estimated.	
ATAM	Approach based on SAAM	Requirement Specification taken into account during the evaluation of the architecture.
SAAMCS	Approach based on SAAM	Requirement Specification taken into account during the evaluation of the architecture.
ESAAMI	Approach based on SAAM	Requirement Specification taken into account during the evaluation of the architecture.
SAAMER	Approach based on SAAM	Requirement Specification taken into account during the evaluation of the architecture.
ALPSM	Not considered	It estimates the required maintenance effort for a software system during architectural design. The maintenance profile is a second output from the method.
PCS Framework	Not considered	Generated Aspect-Oriented Models can be mapped to AspectJ programs, which is a manual process.
DAOP-ADL	Not considered	DAOP-ADL supports traceability in the sense that all the components and aspects identified during the design of the application with CAM are described in XML using the DAOP-ADL language. Additionally, all the information provided with DAOP-ADL is directly used at runtime, closing the "gap" between the design and implementation of component and aspect-based applications
AOGA	Not considered	There is a direct trace link between a crosscutting feature in domain analysis and an aspectual component in domain design. In addition, an architectural aspect is mapped to a set of implementation aspects that refine that aspectual component.
TransSat	Not considered	TranSAT supports the entire software development lifecycle as a iterative evolution process. Concerns are merged together until a system is complete. During iterations the core grows and evolves. TranSAT applies also to the maintenance phase of a software development lifecycle. In this case, development is finished and a core architecture and system exists. Evolution through expanding the architecture with new technical concerns could be considered to be maintenance.
ASAAM	Not considered	Requirement Specification taken into account during the evaluation of the architecture.

Table 4-1: Summary of traceability criterion

With respect to *traceability of artefacts to their source of origin and change*, as shown in Table 4-1, neither the non AO architecture design approaches nor the AO approaches provide support to trace changes in architecture artefacts. Only in SAAM, and the rest of architecture evaluation approaches based on it, possible changes to the architecture are first anticipated and then listed together with an estimation of the cost of performing these changes.

With respect to *traceability between lifecycle artefacts*, most approaches provide some level of traceability to previous phases (e.g., requirements) or later phases (e.g., implementation) of the software lifecycle, though in general they offer only partial information that do not allow a complete trace of artefacts through all the phases of development. Only three of them, DAOP-ADL, AOGA and TranSat explicitly define the architecture design as part of a more complete development process, including other phases of the lifecycle, though not all of them.

The rest of the approaches offer limited and different support for traceability. ADLs, as is the case for ACME, C2, Aesop, Darwin, Rapide and Wright, go from approaches that allow the definition of properties with information related to analysis and design (e.g., ACME), to approaches that provide support to code generation (e.g., C2, Rapide), to approaches that allow to check that resulting implementations conform to the constraints and properties specified during the architecture design (e.g., Darwin, Rapide, Wright).

Traceability in conceptual models such as Model-Driven architecture, Requirement-Driven architecture and Pattern-Driven architecture processes will depend on specific approaches conforming to these conceptual models. In the case of DSAA, which is a specific approach conforming to the conceptual model defined for Product-Line architecture processes, traceability is achieved from requirements to architecture design. This is achieved by using the *reference requirements* as an input to the design of the architecture, where the *reference requirements* include functional requirements, non-functional requirements, design requirements and implementation requirements.

Finally, architecture evaluation approaches, including non AO and AO ones, support some level of traceability from requirements during the evaluation of the architecture, considering requirements as an input to the evaluation process.

4.4.2 Composability

Composability is described in Section 2 as the ability to compose artefacts and consequently to view and understand the complete set of artefacts and their interrelationships, as well as to perceive the system as a whole from the associated artefacts.

Approach	**Features that support Composability**
ACME	ACME *components* and *connectors* are composed in *Systems* – a graph where nodes are components and arcs represent connectors. A *representation map* describes the connections between component ports and connector roles. Acme supports hierarchical descriptions of architecture, permitting any component or connector to be represented by the composition of one or more detailed, lower-level description.
C2	Components linked together by connectors in accordance with a set of style rules.
Aesop	The composition of artefacts (components, connectors, ports and roles) are described by means of configurations, representations and bindings. Hierarchical description of components and connectors is supported.
Darwin	Darwin supports a *bind statement* which is used to tie together components using their ‘provide and require’ statements. The Darwin compiler checks that connections

	are only made between compatible communication objects.
Rapide	Composition of systems from component interfaces is described by an *executable architecture definition language*. The language defines components' synchronisation and communication interconnections in terms of patterns of events.
Wright	Component and connector instances are composed by defining which component's ports are attached to which connector's roles.
UML for Architecture	- Using UML "as is": It does not fully satisfy the structural needs of architectural descriptions. Lack of specialised constructs for modelling architectural artefacts and rules for architectural styles. - Constraining the UML meta-model using UML's built-in extension mechanisms: Using stereotypes and OCL it is possible to enforce the rules of a particular architectural style.
Requirements-driven AD	The Architecture Description is the composition of the following models: business & domain models, use-case models (representing requirements) and analysis & design models.
Domain-driven AD	Architectural design abstractions as a composition of requirement specifications and domain models
DSSA (Product-line Driven AD)	The *application architecture* is an instantiation of a *reference architecture* for a family of products. A *reference architecture* is the composition of a *reference requirements model* and a *domain model*.
Pattern Driven AD	Artefacts are composed according to the architectural style described by a specific *architectural pattern*.
SAAM	Composition of architectural artefacts not applicable in this analysis approach. Single architecture analysis is defined in terms of the analysis of individual scenarios and scenario interaction. Individual scenarios are composed of scenario development and architecture description models.
ATAM	Composition of architectural artefacts not applicable in this analysis approach. Analysis of architecture designs is defined in terms of a composition of elements: high-priority scenarios, attribute-specific questions, architectural approaches, sensitivity points, trade-off points and risks.
SAAMCS	Composition of architectural artefacts not applicable in this analysis approach. Analysis of architecture designs is defined in terms of architecture description, scenario development, categories of complex scenarios, macro and micro architectural description and measurement instruments.
ESAAMI	Composition of architectural artefacts not applicable in this analysis approach. Analysis of architecture designs is defined in terms of reusable architecture & analysis templates, domain-specific analysis templates, architecture description, problem description and requirement statements.
SAAMER	Composition of architectural artefacts not applicable in this analysis approach. Analysis of architecture designs is defined in terms of different types of information, namely, stakeholder information, architecture information, quality information, and scenarios.
ALPSM	Composition of architectural artefacts not applicable in this analysis approach. The analysis approach takes as input the requirement specification, the design of the architecture, expertise from software engineers and historical maintenance data.
PCS Framework	It supports the composition of independent software components along different interaction concerns by means of Aspect-Oriented Constructions. Concerns of multiple dimensions are depicted in an architectural view.
DAOP-ADL	Composition of components among them and weaving of aspects with components (aspects' pointcuts) are expressed in XML. Connections between components are implicitly expressed by the correspondence between their provided and required interfaces. Aspect pointcuts are described in terms of the description of component and aspect interfaces.
AOGA	Architectural description as a composition of aspectual components and non-

	aspectual components. Composition is expressed in UML by means of crosscutting interfaces, normal interfaces and the relationships between them (crosscut relationship)
TransSat	The core architectural model is a composite concern that grows at each evolution of the architecture. Concerns are developed completely independently of the core and then integrated with the core.
ASAAM	It refactors architecture designs to be converted into a composition of architectural components and architectural aspectual scenarios.

Table 4-2 : Summary of composability criterion

With respect to *composability*, Table 4-2 shows that considered architecture design approaches provide some level of composition between the artefacts in the architecture. The main difference among them is in the kind of artefacts that are composed among them.

In the case of ADLs, artefacts are components and connectors for non AO approaches and components, concerns and explicit or implicit connectors for AO approaches. All of them offer composition of these artefacts to describe the complete architecture of a system. Some of them explicitly introduce the concept of *system* as a composition of components and connectors (e.g., ACME, Aesop). In other ADLs composition is expressed in terms of *attachments*, *binding* or *composition constraints* sections where composition is defined in terms of the provided and required interfaces of artefacts or their ports and roles (e.g., DAOP-ADL, Darwin, Wright).

In the case of architecture design processes, artefacts are usually a set of different kinds of models (e.g., business model, domain model, etc.) that are composed among them or taken into account to derive the architecture design model. Finally, in architecture evaluation processes artefacts are different kinds of models, scenarios, existing architecture description, etc. All of them are taken into account during the analysis of candidate architectures.

4.4.3 Evolvability

In Section 2 evolvability is described as the ease of changing the artefacts for an existing design or the addition or removal of a new one.

Approach	Features that support evolvability
ACME	It provides *constraints guidelines* for how the architecture can change over time. New components, connectors and ports can be incorporated into an existing system.
C2	New components and connectors can be incorporated into an existing architectural style. Connections can be modified/adapted when component interfaces and connector ports match.
Aesop	New components and connectors can be incorporated into an existing system. Matching of component interfaces and connector ports needed to modify/adapt the architecture.
Darwin	It supports variation of components, allowing a component to have the potential to define more than one configuration structure, and to be able to defer the choice until the component is instantiated.
Rapide	Architecture can be refined by defining detailed modules for existing components. When an application is "(re)architected" the conformance of new incorporated and existing modules is checked. Connections, features and interfaces are dynamic and can also evolve at runtime, depending on specific runtime parameters.
Wright	New components and connectors can be incorporated to an existing system by attaching the corresponding component's ports with connectors' roles.

UML for Architecture	New UML elements (classes, interfaces, associations, dependencies, etc.) can be added to the UML architectural diagrams to incorporate new architectural artefacts.
Requirements-driven AD	This document describes a conceptual model for requirements-driven AD. Support for evolvability depends on particular requirements-driven AD approaches
Domain-driven AD	This document describes a conceptual model for domain-driven AD. Support for evolvability depends on specific domain-driven AD approaches.
DSSA (Product-line Driven AD)	Application architectures can be extended and/or refined.
Pattern Driven AD	This document describes a conceptual model for pattern-driven AD. Support for evolvability depends on specific domain-driven AD approaches.
SAAM	Possibility of comparing new and existing architecture designs by generating single architecture analysis for the new incorporated architecture designs. Incorporation of new evaluations is reflected in the overall evaluation.
ATAM	Approach based on SAAM
SAAMCS	Approach based on SAAM
ESAAMI	Approach based on SAAM
SAAMER	Approach based on SAAM
ALPSM	New maintenance tasks can be categorised, synthesised and estimated to predict the maintenance effort for such scenario. New maintenance efforts are used.
PCS Framework	Additional perspectives of an existing architecture design can be incorporated to provide additional concerns from one specific viewpoint.
DAOP-ADL	New components, aspects and connections among them can be incorporated into an existing architecture. Only a correspondence between interfaces of the new and existing entities and the definition of new composition constraints is needed.
AOGA	New aspectual components and non-aspectual components can be incorporated into the architecture description by appropriately connecting them by their crosscutting and normal interfaces.
TranSat	It proposes the incremental definition of the software architecture by weaving new architecture plans within a software architecture.
ASAAM	During the analysis of the architecture design scenarios evolve and are categorised into direct scenarios, indirect scenarios, aspectual scenarios and architectural scenarios.

Table 4-3 : Summary of evolvability criterion

With respect to *evolvability* (Table 4-3) and ADLs, all the considered ones support the incorporation of new artefacts into an architecture design, including AO approaches such as DAOP-ADL and AOGA. The only constraint is to ensure that the *connection rules* among the existing artefacts and the new ones are possible (according to existing and new ports, roles, provided interfaces, required interfaces, etc.). Most of them are based on the use of formalisms that allow verifying that the modifications are correct. Additionally, some of them provide explicit support to express dynamic behaviour of the architecture, supporting its dynamic evolution (e.g., Darwin, Rapide, Wright).

With respect to conceptual models such as Model-Driven architecture, Requirement-Driven architecture and Pattern-Driven architecture processes evolvability will also depend on specific approaches conforming to these conceptual models. In the case of DSAA, which is a specific approach conforming to the conceptual model defined for Product-Line architecture processes, evolvability is achieved by extending/refining application architectures.

With respect to architecture evaluation approaches such as SAAM and the rest of approaches based on it, some level of evolvability is achieved by allowing the evaluation of single architecture designs to be incorporated into an existing overall evaluation in an incremental way. In the case of ALPSM, an architecture evaluation approach not based on SAAM, evolvability is achieved by incorporating new maintenance tasks to calculate the maintenance effort for the complete system. ASAAM is an AO architecture evaluation approach where scenarios evolve during the analysis of the architecture.

For the rest of systems not commented on yet, the PCS framework achieves evolvability of the architecture design by incorporating new perspectives of the architecture for one specific "concern" viewpoint. In TranSat the evolution of the architecture is achieved by incorporating new architecture plans.

4.4.4 Scalability

Approach	Features that support scalability
ACME	A semantically extensible language to support complex architectural features, and a rich toolset for architectural analysis and integration of independently developed tools. The simple core set of concepts in ACME can be extended using properties, constraints, types and styles that are appropriate to the context of use. Scalable by supporting a hierarchical model.
C2	No bound on the number of components or connectors that may be attached to a single connector.
Aesop	Scalable by supporting a hierarchical model. The characterisation of architectural styles as specialisations through subtyping can help to develop different projects with different architectural styles. Toolkit for creating an open architectural design environment from a description of a specific architectural style.
Darwin	Scalable by supporting a hierarchical model, tractable and accompanied by a corresponding graphical notation. Increase reusability supporting generic structures and derived composite components (inheritance).
Rapide	Rapide provides *interface services* and the concept of *dual services* as an approach to scalability issues. Services are interfaces within interfaces. They provide a way to structuring interfaces in sub-interfaces.
Wright	Connectors are defined and analysed independent of their actual use, and then later instantiated to describe a particular system, thereby supporting reuse. Support for reusing helps to cope with large complex architecture definitions.
UML for Architecture	The use of UML in this section has been described as used to express a specific architectural design approach or architectural style. Besides the scalability that UML may offer, scalability will depend on the mechanisms defined by the specific architectural design approach.
Requirements-driven AD	This document describes a conceptual model for requirements-driven AD. Support for scalability depends on particular requirements-driven AD approaches
Domain-driven AD	This document describes a conceptual model for domain-driven AD. Support for scalability depends on specific domain-driven AD approaches.
DSSA (Product-line Driven AD)	Several application architecture can be instantiated/refined and/or extending from a *reference architecture*.
Pattern Driven AD	This document describes a conceptual model for pattern-driven AD. Support for scalability depends on specific pattern-driven AD approaches.
SAAM	Scalability is supported by adding new single architecture analysis to the overall evaluation, modifying the weighting according to the values introduced by new analysis.
ATAM	Approach based on SAAM
SAAMCS	Approach based on SAAM
ESAAMI	Approach based on SAAM
SAAMER	Approach based on SAAM
ALPSM	Scalability is supported by adding new maintenance effort scenarios to the overall evaluation.
PCS Framework	Helps deal with software complexity by supporting the composition of independent software components along different interaction concerns.

DAOP-ADL	It provides support to define the architecture of both small and large projects. Supporting tools are provided to cope with the complexity of describing complex and large architectures.
AOGA	Deal with scalability problems using a code generator tool that maps abstractions of the elements in the Agent DSL (XML Schema) to specific compositions of objects and aspects in the agent architecture. *Agent architecture generator* implemented as an Eclipse plug-in.
TransSat	Scalability achieved by the definition of *software architecture pattern*s and by encouraging the reusability of concerns. First, integration of new concerns is performed by the framework. Second, concerns should be generic enough to be reusable in several contexts.
ASAAM	Process supported by the tool environment ASAAM-T, implemented as an Eclipse plug. Tool support helps to cope with the refactoring of large-scale complex architecture designs.

Table 4-4 : Summary of scalability criterion

With respect to scalability (Table 4-4), most approaches cope with it by providing some kind of tool support that helps software architects to use the particular approach in large projects. In the case of ADLs, these tools allow not only to generate code from architecture design but also to analyse and verify that the artefacts and the interconnections among them conform to the properties and constraints explicitly specified by the software architect. Hierarchical specification of artefacts is another approach provided to cope with scalability issues. Support for reusability of artefacts and the definition of patterns are other approaches to help with the complexity of large projects.

5. AO Design

5.1 Introduction

The design activity of a software development process gives a designer an opportunity to reason about a required software system as defined by a set of requirements. This process of reasoning about the system entails consideration of the behaviour necessary for the system to achieve its goals, and a corresponding structure to support that behaviour. For example, in an object-oriented software system, the designer is likely to consider behaviour in terms of (at least) interaction and state diagrams, and structure in terms of (at least) class and object diagrams. The designer may iterate over the behaviour and structure, considering more and more levels of detail over time. The resulting output of the design activity is a set of models that characterise and specify the behaviour and structure of the required system. These models may be at different levels of abstraction depending on the level of detail of the designer's reasoning. Standard software engineering quality measures of the output include the cohesiveness and coupling of the modules described.

Aspect-oriented design (AOD) has the same objectives as any software design activity; to characterise and specify the behaviour and structure of the software system. Its unique contribution to software design relates to extensions to modularity capabilities. Concerns of a software system that are necessarily scattered and tangled in non-AOD approaches can be modularised. Corresponding module cohesiveness is enhanced, and module coupling reduced. An AOD approach will provide design language constructs to support concern modularisation, regardless of whether a concern has an impact on (or crosscuts) other concerns. AOD will also support corresponding specification of concern composition, with due consideration for conflicts or co-operations. Beyond that, the design of each individual modularised concern is likely to mirror standard software design.

An AOD approach is likely to include a *process* and a *language.* An *AOD process* is one that takes as input requirements (be they engineered in an AORE process or otherwise) and produces a design model that may partially realise an architecture. The AOD model produced during the AOD process represents separate concerns and relationships between these concerns. This model is an abstract specification for implementation which can occur on an AOP platform or otherwise.

An AOD *language* is a language that includes constructs that can describe the elements to be represented in design and the relationships that can exist between those elements. In particular for AOD languages, constructs are provided to support the modularisation of concerns, and the specification of concern composition. This includes a means to capture conflict and co-operation specifications.

5.2 Specific criteria

The unique contribution of AOD is its support for enhanced modularisation based on concern. In this report, we therefore examine each AOD approach based on the **level of concern separation**. We also examine the **level of abstraction** at which consideration of "aspects" (or concern modules) is possible

5.2.1 Level of abstraction supported:

A design approach can be thought of as an intermediate translation between requirements and implementation. The level of abstraction is important for an AOD approach as different decisions can be made as design moves from an abstract high-level design model to a concrete low-level. As described in this report, we have encountered various levels of abstraction for the approaches discussed in this section.

Concerns and relationships between concerns expressed at high levels of abstraction are close to requirements and are low in the level of detail required to implement them on specific platforms. Concerns and relationships between concerns expressed at low levels of abstraction contain the details required to implement the concern on particular platforms but may be less naturally mapped to requirements from which the software was developed. Low levels of abstraction make changes to requirements difficult to deal with.

Software engineering quality attributes such as *reuse*, *flexibility* and *comprehensibility* are closely linked with our *level of abstraction* criterion. A design at a high level of abstraction is reusable as it is not likely to be tied to any implementation; it is flexible because it is relatively simple and less resistant to change; and it is easy to understand, as it is not laden with details. As design becomes more concrete and low level, reusability, flexibility and comprehensibility lessen. When choice of a platform becomes constrained, some design decisions may be less feasible on different platforms, reducing reuse and flexibility. As design becomes more concrete, the size and detail of designs also increase, reducing comprehensibility.

There is a conflict between the advantages gained from high level of abstraction and the practicalities of implementing design. On one hand, abstraction provides the higher levels of reusability, flexibility and comprehensibility, but on the other developers need to implement designs and for this they need design to be concrete and mapped directly to a target implementation platform [134].

5.2.2 Level of concern separation:

There are two levels of concern separation supported by existing AOD approaches, *symmetric* and *asymmetric* .[63] In general, the distinction relates to whether an approach includes a means to separate all kinds of concerns, both crosscutting and non-crosscutting (symmetric) or includes a means to just separate crosscutting concerns from the rest of the system (asymmetric).

Some significant benefits of concern separation across the software development lifecycle include traceability, flexibility, comprehensibility, reusability, composability, scalability and evolvability. Symmetric approaches offer a greater opportunity than asymmetric approaches to separate concerns and therefore may provide greater benefits during design. However, many design approaches have followed asymmetric models. In many cases, the asymmetric model has been employed because the AOD approach has emerged from a particular AOP platform. These AOD approaches target the AOP platform for design implementation.

[63] W.H.Harrison, H.L. Ossher, P.L. Tarr. *"Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition"* IBM Research Division, Thomas J. Watson Research Center. RC22685, December 30, 2002.

The difference between symmetric and asymmetric approaches impacts both the design process and design language.

5.2.2.1 AOD Process

The Rational Unified Process (RUP) [135], Feature driven development (FDD) [136], Fondue [137], Catalysis [138], Tropos [139], Kobra [140] and eXtreme Programming (XP)[141] are example of current non-AO software development processes. Many of these processes already separate concerns into concern development processes.
A concern development process is a separate process for developing a concern. The overall process is a composition of separate concern development processes. Table 5-1 presents these processes and the concern development process associated with them.

Process	**Concern development process**
Rational Unified Process	Work flows
eXtreme Programming	Stories
Feature Driven Development	Features
Tropos	Goals
Fondue	Scenarios
Catalysis	Actions
Kobra	Products

Table 5-1 Concern Development Processes

Although these processes separate concerns in terms of a process, each process works on a core system. The system is divided into modular units that can be worked on in complete separation and composed during or at the end of the software development process. During the design phase of the software development lifecycle, a design is created during the development that necessarily scatters and tangles different concerns of the system. Because concerns are not separate, changes in any one concern will impact other related concerns.

Asymmetric AOD processes allow the separation of crosscutting concerns from the core system. Non-crosscutting concerns may be designed in a similar manner to the processes described above. Crosscutting concerns may however be separately designed as aspects. In an asymmetric design process, there are parallel concern development processes - one that creates and expands the core design model and another that creates aspect designs that crosscut that core.

Symmetric AOD processes facilitate separate concern development processes. Each process creates separate software design modules that realise a system in design. There is no core system. There is one type of concern development process. Unlike the existing non-AO processes these processes do not work on creating or expanding a core system. Symmetric concern development processes allow the creation of separate designs that realise crosscutting and non-crosscutting concerns.

5.2.2.2 AOD Language

An AOD language consists of some way to specify aspects; some way to specify how aspects are to be composed; and a set of well defined composition semantics to describe the details of how aspects are to be integrated.

Symmetric and asymmetric AOD languages are similar in that they both provide a means of separately specifying crosscutting concerns as aspects, specifying aspect integration and composition semantics for aspect. Symmetric AOD languages further provide a means of modularising non-crosscutting concerns. They enable the specification of how these non-crosscutting concerns are integrated with other non-crosscutting software modules as well as aspects, expanding the composition semantics accordingly.

5.3 Non-AO Approaches

The Unified Modelling Language (UML) is the OMG standard design language for OO design. It is the most widely used object-oriented design language, and has effectively precluded the use of any other language except in exceptional circumstances. The UML has also been the basis from which most AO design languages have been developed, apparently sensible where aspect-oriented design is primarily an enhancement to object-oriented design. We therefore limit our attention to non-AO approaches to a discussion on the UML.

5.3.1 UML

The UML provides design models that are adequate for capturing and representing problems in the object-oriented paradigm. The UML allows particular concerns to be represented through different OO design models. Views of the structure and behaviour of concerns can be separated.

5.3.1.1 UML Artefacts

In this section, we look at elements of standard UML, which will provide a basis for the discussion of many of the AOD approaches described in this report.

5.3.1.1.1 UML architecture

UML is the graphical notation that defines the semantics of the object meta-model, and defines a notation for specifying and communicating object structure, behaviour, state and interaction. In its meta-model architecture, UML supports extension mechanisms that allow tailoring it to fit the needs of a specific domain such as AOD.

The Object Management Group (OMG) defines UML as a four-layer architecture (see Figure 5-1). The bottom level of Figure 5-1 defines the meta-meta-model layer, which is defined by the Meta Object Facility (MOF). The UML meta-model specifies the modelling language; it is defined and standardized on top of the MOF in the level second from bottom. Profiles (which are further examined later) are defined at this level, they specialize the meta-model for particular domains that are being modelled. The level labelled model defines the model layer, and the user object layer defines instances of the model at the top level [142].

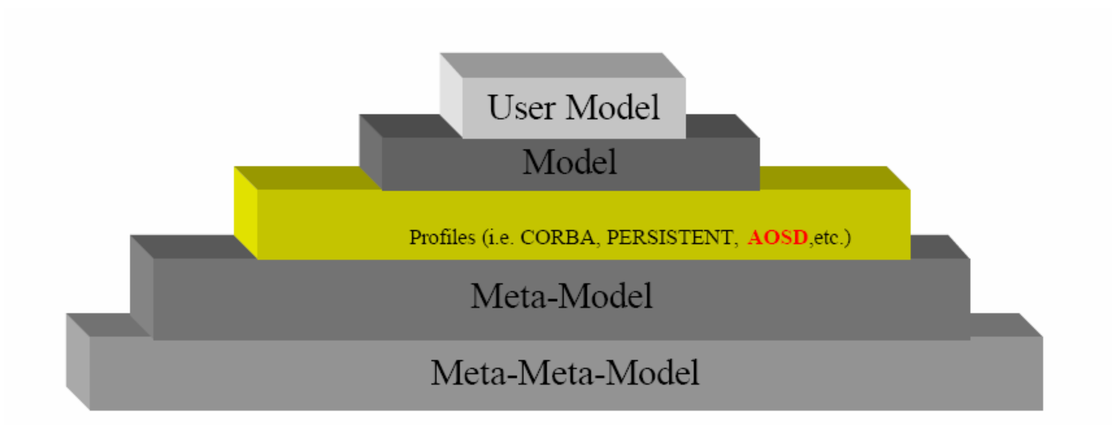


Figure 5-1 UML Architecture [142]

5.3.1.1.2 Standard UML

Standard UML has been used to support AOD. At this level standard OO diagrams are used to represent AO base systems. This generally is semantically limiting as the standard OO diagrams have not been designed to represent the concepts and relationships that embody the AO paradigm. Although the use of standard UML diagrams requires no extension of UML to support AOD it does not express the semantics required for AOD.

5.3.1.1.2.1 UML Extension mechanisms

UML extension mechanisms allow a standardised method to augment UML elements with new properties or semantics. Adding properties or tagging values can be used to add information to elements. New semantics or constraints can be specified for a model element, refining the concept the model represents. New UML elements can also be composed of existing elements. New elements, called stereotypes, have the same structure to the elements from which they are derived but differ semantically and may require additional tagged values. Stereotypes can be used to indicate the difference between elements with identical structure. Due to a certain amount of perceived similarity between the OO and AO paradigm, many approaches have used these extensions to create new AOD elements from OOD elements. Extensions on their own do not provide a consistent integrated tool set for AO designers to create large-scale designs.

5.3.1.1.2.2 UML profile

A UML Profile is a predefined set of extension mechanisms. A UML Profile allows the specialisation of the existing UML design elements for a certain domain. A UML Profile enables the expression of the semantics of the domain systems using a well-defined set of extensions. A profile does not extend the UML meta-model. Profiles are extensions of UML models and as such the principles of UML must not be broken, but must be extended. Profiles provide that consistent integrated tool set.

5.3.1.1.2.3 Meta-model

Many have found that the extension mechanisms available are not enough to support AOD. The extension mechanisms allow the extension of OO semantics. It may be argued that an AOD language cannot be specified in terms of OO semantics and the extensions thereof. The approaches that take this view extend the UML language itself and alter the UML meta-model. This allows the complete definition of AO semantics without adhering to the constraining features of the OO paradigm.

UML is a standard language. The use of standard extension mechanisms either on their own or grouped in profiles respects the standard. UML is a common language. By altering the standard this makes any AOD language conceived in this way non-standard, however semantically fitting.

5.3.1.1.2.4 UML 1.X diagrams

Diagrams	Scope & Abstraction	Models
Use Case Diagrams	Inter system	Behaviour
Component Diagrams	Inter component	Structure
Class Diagrams	Inter class	Structure
Object Diagrams	Inter object	Structure
Collaboration Diagrams	Inter object	Behaviour
Sequence Diagrams	Inter class	Behaviour
Activity Diagrams	Inter class	Behaviour
State Diagrams	Intra class	Behaviour

Table 5-2 UML Diagrams

UML 1.X provides several views to model the static and dynamic behaviour of a software system. Table 5-2 identifies each of the diagrams provided by the UML. For each we identify what the diagram models, the diagrams scope and also the level of abstraction is provided by the diagram.

As illustrated in Table 5-2 we consider UML diagrams [143] to model dependency, structure and behaviour. What is modelled is very much linked with the scope and level of abstraction that a particular diagram provides.

Inter system indicates that the scope is a system level scope where a system is being modelled at a very high level of abstraction. At this level of abstraction the overall systems behaviour is modelled. Use case diagrams model the high level structure and behaviour of a system.

Inter component indicates that the scope is a component level scope where a system is being modelled in terms of the components that is contains .A component diagram exposes the structure of a system by indicating the components in a system and the relationship between components.

Inter class indicates that the scope is a class level scope where the system is being modelled in terms of class. At this level of abstraction structure and behaviour can be modelled. Classes are used to compose features. Class diagrams model class structure and the structural relationship between classes. Sequence and activity diagrams model behaviour across the class structure.

Intra class indicates that the scope is a class and models the states that instances of a class can be in and the conditions in which state transitions occur. State diagrams model the state and state transitions of a class.

Inter object indicates that the scope is an object level scope where the system is being modelled in terms of objects. Models at this level depict runtime structure and interactions based on behaviour and state. Object diagrams model runtime structure and collaboration diagrams model runtime behaviour.

5.3.1.1.2.5 UML 2.0

UML 2.0 has increased the scope to separate concerns in UML by providing new opportunities to separate concerns into separate diagrams and new constructs that can represent designation. The significant improvements in UML2.0 include ([144]):

Diagrams	Improvement	AOD impact
Class & Component	New concepts for describing the internal architectural structure of Classes, Components and Collaborations by means of Part, Connector and Port.	Better separation of components may aid aspect representation [145]
State machines	Introduction of inheritance of behaviour in state machines and encapsulation of sub machines through use of entry and exit points.	Can aid modelling crosscutting behaviour
Activity diagrams	A redefinition of Activity diagrams in which Petri net flow semantics are used instead of state machines and extension points are introduced.	Can aid modelling crosscutting behaviour [146, 147]

Table 5-3: UML 2.0 Diagrams

5.3.2 UML and AOD

Standard UML was not designed to support the kinds of concern separation that are the focus of AOSD. Nonetheless, it provides the basis for object-oriented design from which many AOD approaches extend. It is therefore not a surprise that many of the AOD approaches discussed in this report have used or extended UML to facilitate AOD in different ways. Different approaches range from extending the UML meta-model itself, to using UML's standard extension mechanisms. Table 5-4 gives a summary for each of the AOD approaches in this report.

Approach	Language	
	UML	Standard/Extensions/Profile/Meta-model
Theme	Yes	Meta-model
AODM	Yes	Extensions
AAM	Yes	Extensions
CoCompose	No	n/a
SUP	Yes	Profile
AML	Yes	Extensions
Concern Modelling	No	n/a
TranSAT	Yes	Extension
AOCE	Yes	Meta-model
UFA	Yes	Meta-model
ADT	Yes	Extension
UXF/a	Yes	Meta-model
IDAM	Yes	Meta-model
AVA	Yes	Extension
Component Views	Yes	Meta-model
Meta Models	Yes	Meta-model
AOSDUC	Yes	Meta-model (UML 2.0)

CAM/DAOP	Yes	Profile
Activity	Yes	Profile (UML 2.0)
UMLAUT	Yes	Meta-model

Table 5-4: UML & AOD

In [148] it is noted that the choice between possible options is based on the level of concerns separation and AOD language targeted. Beyond concern separation, there are other rationales for choosing each [149]. Meta-model extension is suggested if the AO concepts are well defined, stable and not subject to transfer or composition with other domains. UML extensions and profiles are suggested where the aspect domain is not subject to consensus, subject to change and evolution, subject to transfer or composition with other domains.

Tool support has become very important to designers. UML based tools are built on the meta-model. By changing the meta-model new corresponding extensions to UML tools must be made. The use of profiles for extension means that no extensions for tool support are required.

5.3.2.1.1 OCL and AOD

The Object Constraint Language (OCL) is a subset of the UML that allows software developers to write constraints and queries over object models. OCL enables the description of expressions and constraints on models and other modelling artefacts. An expression is an indication or specification of a value. A constraint is a restriction on one or more values of a model or system.

In terms of AOD, OCL has been used to constrain the selection of join points in design models. As such OCL is mainly used in the specification of model integration. Current AOP approaches are supporting more complex integration specifications to identify join points. To keep up with AOP developments this complexity must be mirrored in an AOD language.

OCL is a non-graphical language. As such it is not intuitive to the designer. In [150] [151] a graphical language called Join point designation diagrams that represents OCL constraints are visualised such that the integration complexity they expect from AOP can be expressed in a natural manner at design.

OCL has also been used to define the constraints for aspect models as well as defining the constraining composition. In terms of constraining aspect models, OCL has been used to aid in the ensuring the well-formedness of aspect specifications and integration specification. In terms of composition, OCL can be used to define the composition semantics for an AOD language.

5.3.2.1.2 Parameterised templates

At the core of all AOD approaches (symmetric and asymmetric) is a requirement to represent crosscutting relationships between models. Many UML-based approaches achieve this through the utilisation of parameterised templates.

Parameterised templates allow the creation of abstract design models. A model is abstract in this sense in that it reasons about abstract elements relative to the concrete elements within the model. The abstract element is a parameter (i.e., template) that can be replaced with a concrete element. This replacement generates a new diagram where the replacing concrete element gains the relationships, attributes and constraints associated with the parameter being replaced.

This is one mechanism for representing crosscutting. A crosscutting relationship is one where one model alters a number of different models. As such parameterised templates support the specification of crosscutting.

5.3.2.1.3 AOD and UML Diagrams

As we have discussed previously, AOD has the same objectives as any software design activity; to characterise and specify the behaviour and structure of the software system. Its unique contribution to software design relates to extensions to modularity capabilities. AOD must therefore provide language constructs and a process to handle the structural and behavioural extensions required to manage the separation and composition of the kinds of concerns supported by the paradigm. Table 5-5(a) identifies the diagrams that have been used to express aspect structure and the approaches that these diagrams have been used in. Table 5-6(b) identifies the diagrams that have been used to express aspect behaviour and the approaches that these diagrams have been used in.

	Use case	**Package**	**Component**	**Class**
Theme		Structure		Structure
AODM				Structure
AAM		Structure		Structure
SUP				Structure
AML		Structure		
TranSAT				Structure
AOCE			Structure	Structure
UFA		Structure		
ADT				Structure
UXF/a				Structure
IDAM				Structure
AVA		Structure		Structure
Component Views			Structure	
Meta Models		Structure		Structure
AOSDUC	Structure	Structure	Structure	Structure
CAM/DAOP			Structure	Structure
Activity **UMLAUT**				Structure

Table 5-5 (a) Aspect Structure in UML Diagrams

Structure and behaviour are described under the headings of aspect specification, crosscutting specification, integration specification and composition semantics. It is interesting to note that the choice of diagram used in the AOD approach is based on the level of separation and abstraction that the AOD language supports.

	Use case	Sequence	Communication	Activity	State
Theme		Behaviour			
AODM	Behaviour	Behaviour			
AAM			Behaviour		
SUP					Behaviour
AML					
TranSAT					
AOCE			Behaviour		
UFA					
ADT					Behaviour
UXF/a					
IDAM					
AVA		Behaviour	Behaviour		Behaviour
Component Views					
Meta Models					
AOSDUC	Behaviour	Behaviour	Behaviour	Behaviour	Behaviour
CAM/DAOP				Behaviour	Behaviour
Activity UMLAUT		Behaviour		Behaviour	Behaviour

Table 5-6 (b) Aspect Behaviour in UML Diagrams

5.4 AO Approaches

5.4.1 Aspect-Oriented Design Modelling (AODM)

5.4.1.1 AODM Method

The Aspect-Oriented Design Modelling [152] (AODM) approach extends standard UML with aspect-oriented concepts. The AODM UML extension was originally defined to support the aspect-oriented concepts of the AspectJ implementation language. However, AODM has evolved to become more generic and now supports other asymmetric AO Programming approaches (such as composition filters and adaptive programming) [153] [154]. Yet, symmetric AO implementation techniques are not naturally supported by AODM.

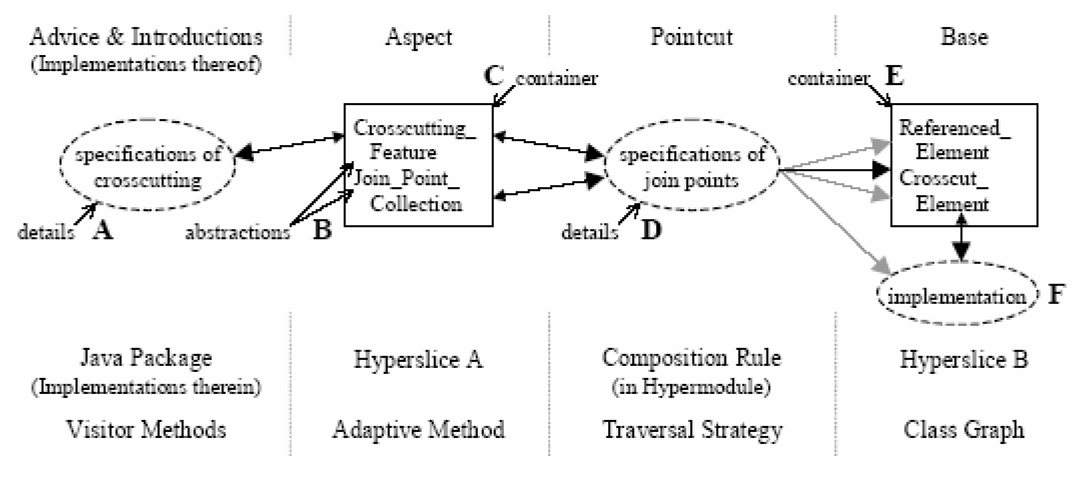


Figure 5-2 Aspect Oriented Design Issues [153]

5.4.1.2 AODM Artefacts

5.4.1.2.1 AOD Language

Since AODM was originally designed to model AspectJ-style AOP, its design elements are still heavily related to AspectJ. We rely on AspectJ terminology when discussing AODM even when discussing design models that target various [150] [153, 154] AOP platforms.

5.4.1.2.2 Specification of Aspects

In AODM, aspects are represented as classes with the <<aspect>> stereotype (see Figure 5-4 and Figure 5-7). This was adopted because of the structural similarities between aspects and classes. Like classes, aspects act as containers and namespaces for attributes, operations, pointcuts, advice and intertype declarations. Aspects can also engage in the same association and generalization relationships as classes .

Aspects differ from classes in their instantiation and inheritance mechanisms. In AspectJ, aspect declarations can contain instantiation clauses that specify the way in which the aspect should be instantiated. Child aspects inherit all features from their parent aspects but only abstract pointcuts and java operations can be overridden. This stereotype augments the meta-class with some additional meta-attributes to hold the instantiation clause, and a Boolean expression to specify whether the aspect is privileged or not.

5.4.1.2.3 Specifications of crosscutting

AODM supports the specification of behavioural and structural crosscutting. Crosscutting structure is expressed in class diagrams within a parameterized template collaboration diagrams.

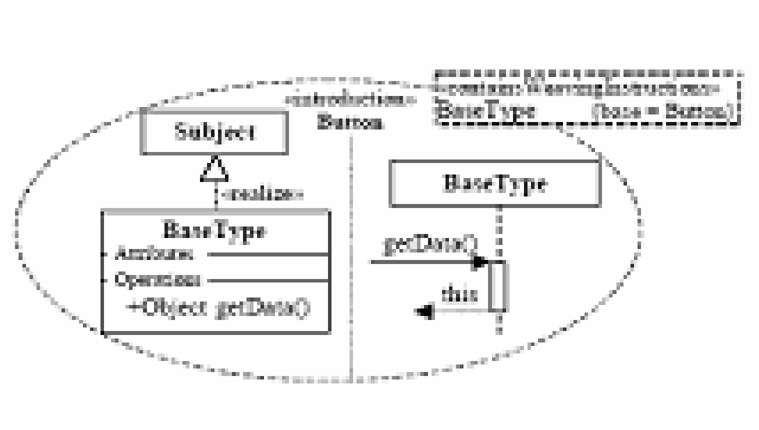


Figure 5-3 Structural Crosscutting [152]

Figure 5-3 illustrates an example of structural crosscutting. Crosscutting structure is captured in a parameterised class and partial sequence diagrams. Parameterisation is used to represent crosscutting. The type(s) to be crosscut are represented as parameter(s) to the template. The crosscutting structure is applied to this parameter. The concrete structural elements to be crosscut are applied to the template as arguments. Class and sequence diagrams dictate the manner in which integration occurs.

In Figure 5-3 all arguments that match the `BaseType` parameter are extended by the `Subject` type. Any type that is extended in this way exposes a `getData()` signature which can be invoked.

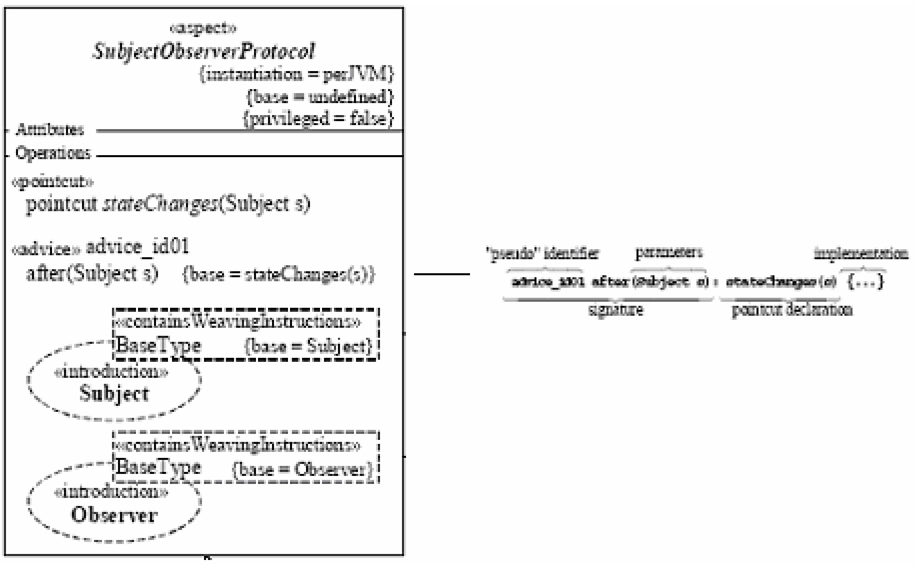


Figure 5-4 Advice AODM [152]

As illustrated in Figure 5-4 behavioural crosscutting is represented as an operation with an <<advice>> stereotype. Advice operations, like standard operations, have a signature and are semantically similar to standard UML operations. Figure 5-4 also reveals the closeness between AODM and AspectJ notations for advice, yet there are some semantic differences between these two.

Firstly, AspectJ advice is not uniquely identifiable. This conflicts with the UML rules that operations of the same classifier must not have the same signature. AODM resolves this problem by using "pseudo" identifiers. Secondly, advice in AspectJ cannot be overridden, as it has no unique identifier. Finally, AspectJ advice signatures reference pointcuts. The operations of the <<advice>> stereotype must be implemented by methods of a special stereotype that has an additional property named "base" to hold the pointcut declaration.

The semantic value that the <<advice>> stereotype in AODM adds to the operation is not immediately apparent. A designer can reason about <<advice>> only if he is aware of the AODM-AspectJ relationship and fully understands the underlying semantics of the AspectJ advice.

Additionally, the link between AODM and AspectJ is broken by using stereotyped operations: use of pseudonyms to identify advice does not match the AspectJ implementation of advice. This highlights the fact that the AOP languages constructs do not always naturally match the UML elements extensions to capture the additional characteristics of the construct.

5.4.1.2.4 Integration specification

AODM supports the specification of structural and behavioural integration through integration of crosscutting design models. Structural crosscutting affects the type structure of a given design model and can occur at some location in a target class hierarchy. Behavioural crosscutting affects the model's behavioural specifications and occurs at some joinpoint in the execution specification.

UML classifiers from within a target class hierarchy may identify points to be structurally altered.

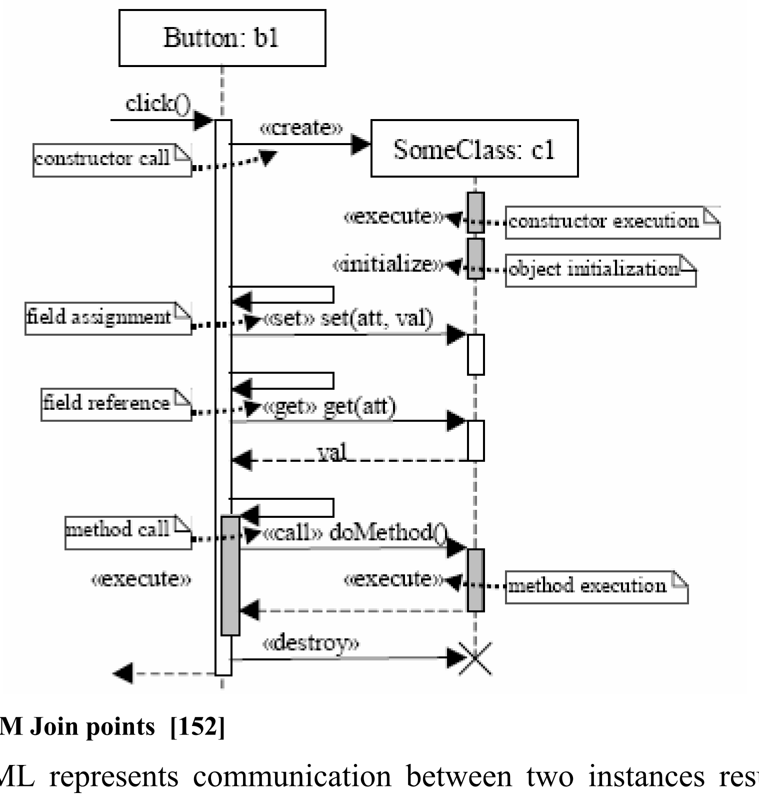


Figure 5-5 AODM Join points [152]

A Link in UML represents communication between two instances resulting from a particular action. As illustrated in Figure 5-5, AODM represents join points as UML links with stereotyped "pseudo" operations. The stereotyping is required as UML links imply a delegation of program control, but some joinpoints (e.g. field access) do not delegate control. These joinpoints then cannot be modelled within the standard UML and must be represented as pseudo operations. There are also cases such as object instantiation, where initialization only occurs after a constructor call. Here one link is used to represent the join points. Interaction diagrams are used to represent the order in which control passes between these join points. The type of join point being modelled is indicated by special stereotypes. The <<execution>> stereotype, for example, indicates an execution join point that occurs in the context of a method invocation.

Differing AOP implementations apply different mechanisms to separate concerns. AspectJ supports behavioural and structural crosscutting, composition filters support behavioural crosscutting, and adaptive programming supports structural crosscutting. The Hyper/J-SOP model differs from AOP models in that AOP models support the augmentation of a single model where as the Hyper/J model supports the integration of multiple models. In the Hyper/J model, crosscutting support is identified as structural because the primary concern is the integration of type hierarchies.

AODM provides design models that can be applied to the AOP models that separate concerns models from a single set of base modes. However, due to the differences in the possible integration between multiple models and the integration between a base and crosscutting model, AODM cannot express all possible integration strategies of Hyper/J.

In the following sections we will look at the AODM design models used to specify structural and behavioural crosscutting.

Structural crosscutting is termed "intertype declarations" in ApectJ. Intertype declarations are used to insert members and relationships into the base structure. Intertype declarations are represented as templates in AODM. These templates are parameterized model elements that are used to generate other model elements by binding its template parameters to actual arguments.

As illustrated in Figure 5-3, a template of the <<introduction>> stereotype is used to capture the semantics of inter-type declarations [152]. Templates are not useable directly in the model design - arguments must be bound to them in order to be used in design models. UML's well-formedness rules allow at most one client model element to participate in one binding. This does not match the AspectJ weaving model where a class can be crosscut by many inter-type declarations. Inter-type declarations in AspectJ are always bound to a fixed number of actual base classes. Template parameters of an <<introduction>> stereotyped collaboration template are required to be of a special stereotype <<containsWeavingInstruction>>. This stereotype augments the template parameter with a meta-attribute named "base" to hold the type pattern that specifies the actual base classes to be crosscut.

In more recent work however there are some slight variations. Although not much detail has been provided [154] [151], the main difference seems to be in usage of the <<join point>> stereotype to indicate structural crosscutting. It remains unresolved if or how this changes the original AODM representation of specifying structural crosscutting.

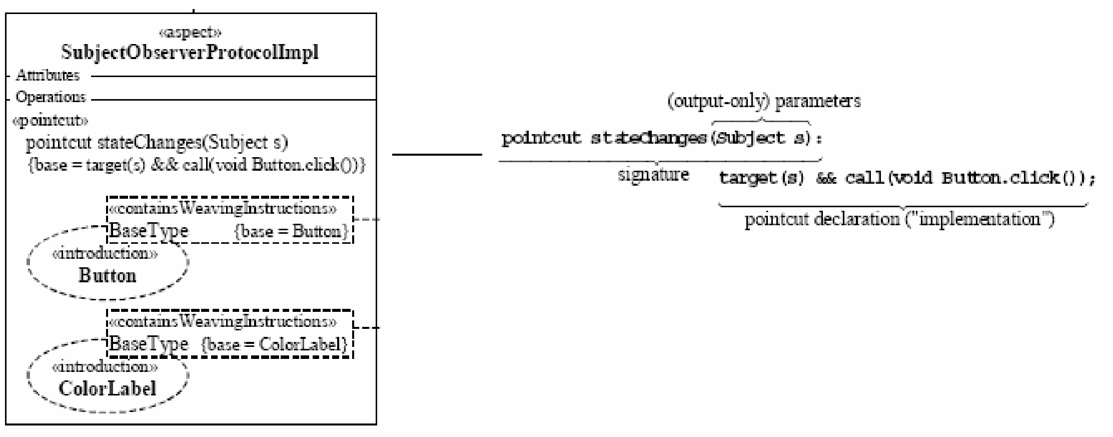

«aspect»
SubjectObserverProtocolImpl
Attributes
Operations
«pointcut»
pointcut stateChanges(Subject s)
{base = target(s) && call(void Button.click())}
«containsWeavingInstructions»
BaseType {base = Button}
«introduction»
Button
«containsWeavingInstructions»
BaseType {base = ColorLabel}
«introduction»
ColorLabel
(output-only) parameters
pointcut stateChanges(Subject s):
signature
target(s) && call(void Button.click());
pointcut declaration ("implementation")

Figure 5-6 Pointcut AODM [152]

In earlier work [152] pointcuts are represented as operations of a special stereotype <<pointcut>>. As illustrated in Figure 5-6, the <<pointcut>> stereotype captures the AspectJ pointcut semantics. Figure 5-6 also shows the corresponding pointcut expressed in the AspectJ language.

There is also a graphical method to represent pointcuts in a more abstract manner avoiding the use of AspectJ specific join point designators. Join point designation diagrams [150, 151] [154] represent behavioural and structural crosscutting in a language independent manner while retaining the complex semantics typically found only in implementation languages. As illustrated in Figure 5-7 join point designation diagrams are reusable templates that specify join points in a constrained manner.

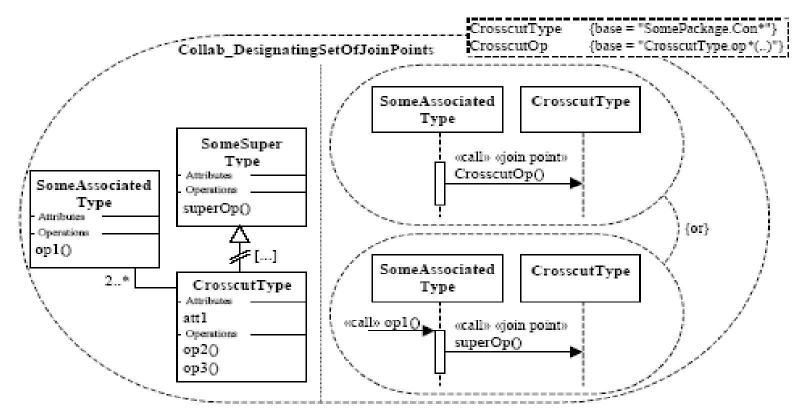


Figure 5-7 Join point designation diagram [154]

Figure 5-7 is an example of a join point designation diagram. This diagram illustrates the graphical specification of a pointcut. On the right side of the diagram are the parameterised sequence diagrams that describe a set of join points. There are two sequence diagrams joined through an "or" association. Each sequence diagram represents a primitive pointcut. The "or" association between these two internal diagrams represents the composition of primitive pointcuts. The left side of the diagram is a graphical constraint language that declares the structural requirements that must be met by arguments that are used to fulfil the right hand side of the diagram.

This graphical constraint language is based on OCL and is described in [151] [150]. As illustrated here in Figure 5-9 a complex join point selection criteria is provided. Constrains are expressed on instances and classes. The constraints represent static and dynamic join point selection constraints. Class and instance based constraints are represented in a class like structure. An example of a constraint representation is given in Figure 5-8. An example of a static constraint is on the classifier name Con* means any type that is to match this constraint must be begin with “Con”. An example of a dynamic constraint is where the multiplicity range of an attribute is set. An instance will match this constraint if it has an attribute att2 that is an integer value between 2 and 100.

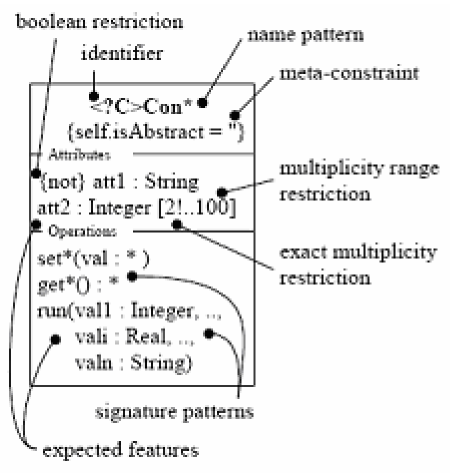


Figure 5-8 UML –classifier selection [151]

Relationships constraints can also be represented. These again can be static or dynamic. There are four types of constraints: association, generalization, specialization and flow of control. Association constraints impose statically constraint relationships between types and dynamically constrains the multiplicity of relationships between instances. Generalisation and specialisation constraints constrain a type in terms of type hierarchies. Named *message selection criteria* constrain join point identification based on the flow of control between types. Message selection constraints are represented in diagrams similar to sequence diagrams.

5.4.1.2.5 Composition Semantics

Detailed composition semantics are not provided for AODM in the literature. Due to its close ties with AspectJ we infer that the composition semantics followed by AODM are very similar to AspectJ.

AODM supports the representation of composition semantics through two diagrams. The first diagram, illustrated in Figure 5-9, identifies the join points that are crosscut by an aspect and crosscutting element in that aspect that will actually affect that join point. The second diagram represents an actual join point and specifies the composition at that join point.

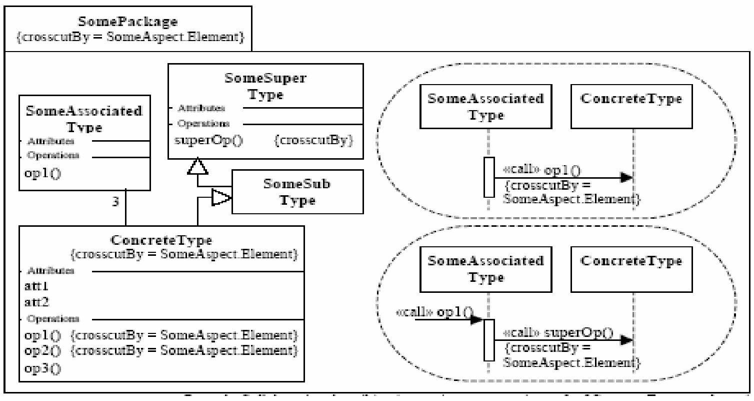

SomePackage {crosscutBy = SomeAspect.Element}
SomeAssociated Type
op1()
SomeSuper Type
superOp() {crosscutBy}
SomeSub Type
ConcreteType {crosscutBy = SomeAspect.Element}
att1
att2
op1() {crosscutBy = SomeAspect.Element}
op2() {crosscutBy = SomeAspect.Element}
op3()
«call» op1() {crosscutBy = SomeAspect.Element}
«call» superOp() {crosscutBy = SomeAspect.Element}

Figure 5-9 Join Point Indication Diagram [154]

The points that are crosscut are captured within Class Diagrams elements for structural crosscutting, and sequence diagrams links for behavioural crosscutting, by augmenting these with a "crosscutBy" property. This property references a crosscutting element within an Aspect. This indicates that an element augmented with the "crosscutBy" property is composed with the crosscutting element named by the "crosscutBy" property.

This diagram provides an overview of all the join points identified by a join point designation diagram. It also shows the advice that will crosscut these join points.

The second diagram represents an actual join point and the composition at that join point is, in fact, a number of diagrams. There are two options provided by AODM for the representation of per-join point composition. The first option is to present composition in a number of partial sequence diagrams. The second is to represent composition through use cases.

An example of the first option is represented in Figure 5-13. To show how, and where, advice affects the base classes an interaction diagram is spit. Splitting occurs at a particular join point. The join point is indicated by a stereotyped link. The serotype specifies the join point type. The advice types supported include before, after or around advice. The advice is recomposed into a new sequence diagram that is made up of the original join point sequence with the advice included.

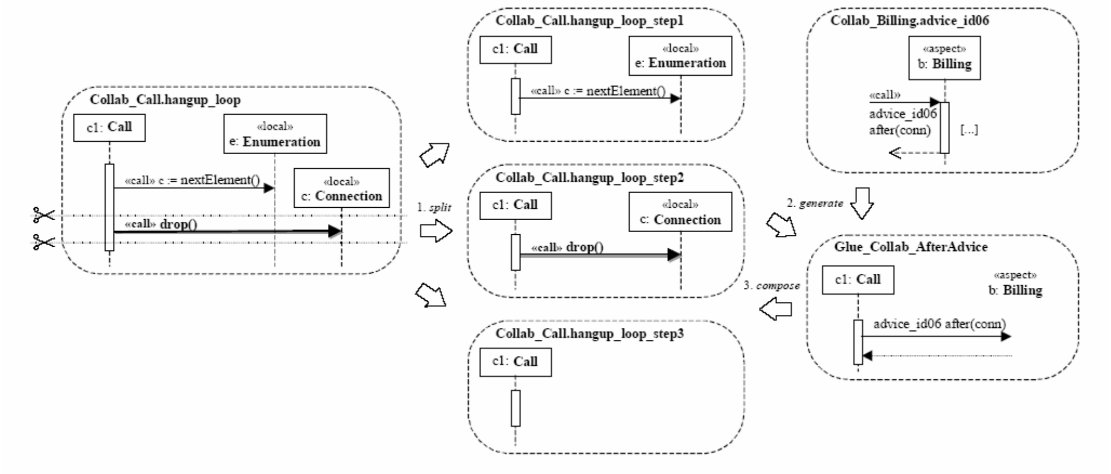


Figure 5-10 Join point Composition [155]

Figure 5-10 shows how a join point can be split into three sequence diagrams to represent the before, around and after points at the join point at which advice can be injected. The advice to be injected at the join point is represented in the sequence diagram in the top right of Figure 5-10. This advice is declared as after advice. The join point split diagram is combined with the advice diagram to create a new diagram (see lower right side of Figure 5-10) which represents the composed join point and advice.

An example of the second option to represent composition is seen in Figure 5-11 and Figure 5-12. Here a use case is used to represent some behaviour. This use case can then be split into smaller use cases via a refinement relationship. Use cases may also include behaviour from other use cases. A use case may augment another by using extension. As such a join point can be refined into before, around and after. Composition of advice and base behaviour can be depicted by refining the join point use case and using the include relationship. In the case of before and after advice or extends in the case of around advice to form a new woven use case that represents the join point composed with advice.

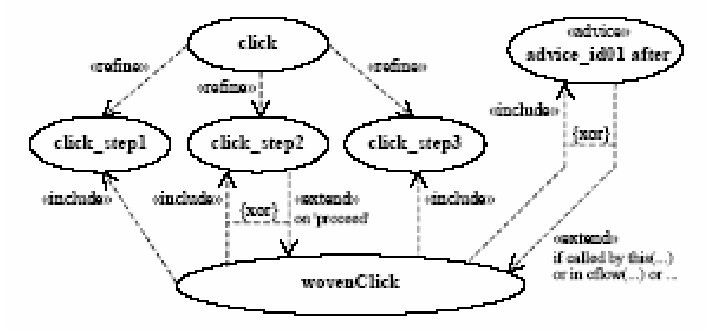


Figure 5-11 Weaving Advice [152]

Figure 5-11demonstrates the weaving at a join point and a piece of advice related to that join point. The join point use case is used to represent a join point. This is again split into three separate use cases. These represent the points at which this join point can be crosscut. The crosscutting behaviour that is to be woven at this join point is also represented as a use case. The actual composition of advice and behaviour at the join point is represented by the wovenClick use case. The association between the elements to be composed and the actual composition representation is an <<include>>.

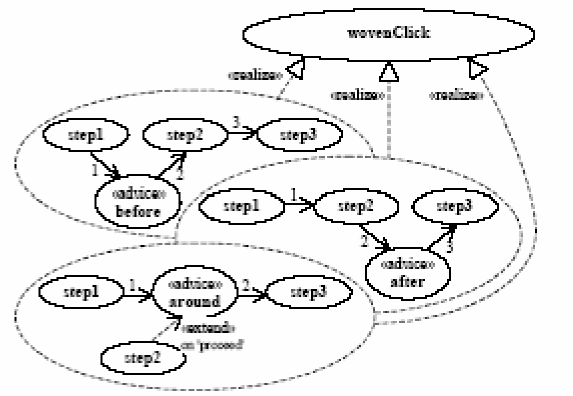


Figure 5-12 Weave Order [152]

Figure 5-12 extends the example presented in Figure 5-11. In Figure 5-11 we see that the actual composition can be visualised. There are the possibilities for advice injection, before, after and around the join point. Each possibility is represented in the figure as a separate diagram through associated with the composition.

The weaving of inter-type declarations is represented in use case diagrams. The aspect that contains the inter-type declarations is represented as a use case. The inter-type declaration is firstly refined from the aspect use case, into use cases to represent the actual features that are to be introduced into the base classes. These use cases are then included in a use case that represents a woven entity. In Figure 5-13 we can see that the composition is represented as use cases. These are related to the base and crosscutting elements that are composed through the <<include>> association.

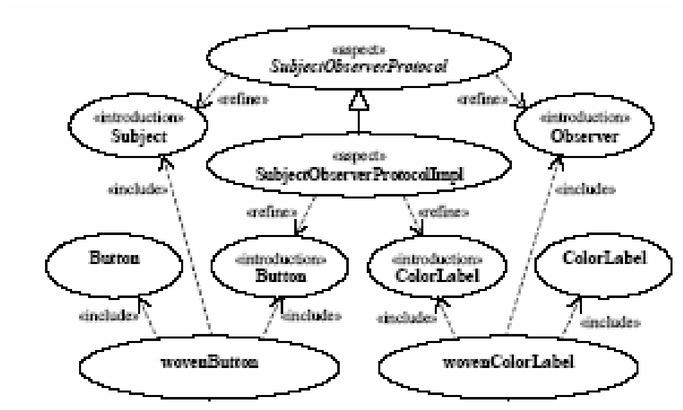


Figure 5-13: AODM TP 2002 Weaving intros [152]

5.4.2 Theme/UML

5.4.2.1 Theme/UML Method

Theme [55] is an analysis and design approach that supports the separation of concerns for analysis and design phases of software lifecycle. The Theme approach also provides a UML based AOD language called Theme/UML which extends the UML meta-model.

The Theme approach expresses concerns in conceptual and design constructs called themes. Themes are more general than aspects, and more closely encompass concerns with relation to the symmetric separation. Any concern, whether crosscutting or not, may be encapsulated in a theme. [156].

5.4.2.2 Theme Artefacts

5.4.2.2.1 AOD Language

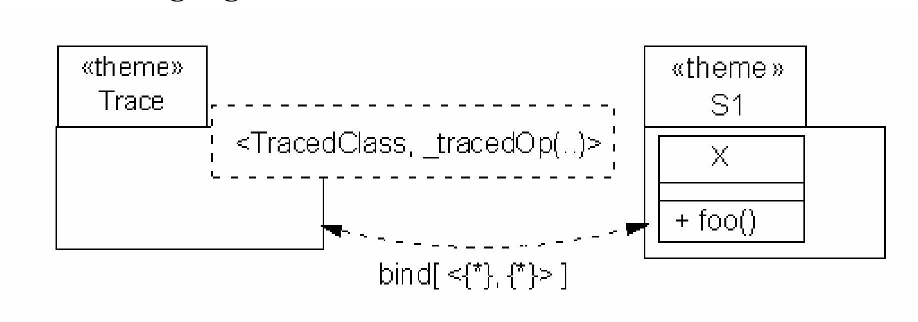


Figure 5-14 Themes and Theme Integration [157]

5.4.2.2.2 Specification of Aspects

Being a symmetric language, Theme/UML supports the representation of base and aspect themes. Base and aspect themes are specified in packages. Theme packages are identified as themes through the stereotype <<theme>>, as shown in Figure 5-14. Base and aspect themes contain class diagrams that represent aspect structure. The class diagram within the theme package represents the design concepts that are required to realize the requirements related to the theme.

An aspect theme differs from a base theme in that it is a parameterized template package. Figure 5-15 is an example an of aspect theme. The dotted box in the top right hand side of the diagram specifies the parameters to the template. The parameters represent the join points that the theme crosscuts. The parameter, in this example, is a Type and method and the parameter is represented as follows <TracedClass _tracedOp(..)>. This example shows the design of crosscutting tracing functionality. The structural relationship between the parameter and the crosscutting elements encapsulated within the theme described in the class diagram. The behavioural relationship between the parameter and the crosscutting elements is encapsulated within the theme described in the sequence diagram.

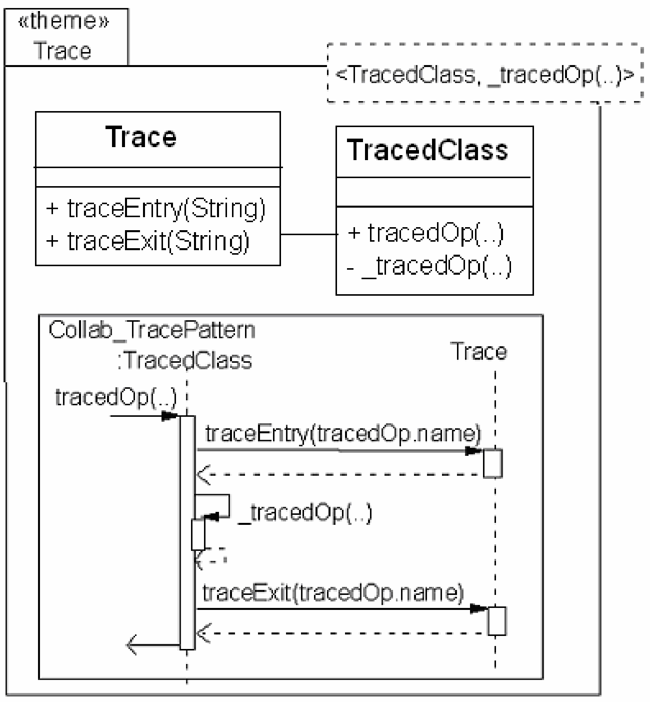


Figure 5-15 Aspect Theme [157]

5.4.2.2.3 Specification of Crosscutting

Both aspect and base themes support a degree of structural crosscutting due to the possibility of domain concept overlap between themes. Aspect themes support behavioural crosscutting as well as structural crosscutting.

At design, themes are partial views of requirements. Requirements express all of the domain concepts that must be modelled to create a system. An overlap between themes occurs when the requirements for different themes partially describe a domain concept. At design, this overlapping is seen in classes in different themes that represent the same domain concept. These classes have members, methods and attributes, as well as relationships with other classes in a theme. Methods and attributes can overlap as can relationships. Overlapping of class members and inter-class relationships is less common. During composition, those overlapping classes are converged into full models. Due to possible overlap between themes, all themes are prone to implicit structural crosscutting. Overlap is relative to other themes and hence, structural crosscutting (in this sense) is too.

Aspect themes provide more scope for structural crosscutting. Returning to the example in Figure 5-15, and in particular the class diagram, structural crosscutting can be explicitly specified in this class diagram. As discussed in Section 4.3.1, aspects are parameterised templates. The parameters represent all join points at which the aspect is to crosscut other themes. A class that represents the parameter type is included in the class diagram. This class is related to the crosscutting classes. Structural crosscutting is specified in the relationships that are expressed between the parameter class and aspect themes classes.

Only aspect classes support behavioural crosscutting. Behavioural crosscutting is specified in a sequence diagram. Figure 5-15 depicts an aspect theme. This aspect theme contains a class and sequence diagram that specifies crosscutting behaviour. The parameter that is specified as a TracedClass type and _tracedOP(..) method. In the class diagram, TracedClass is represented as a class with two methods tracedOp(..) and _tracedOP(..).tracedOp(..) represents the composed method or output and _tracedOP(..) represents the parameter method or input. Any method that replaces _tracedOP(..) is specified in the sequence diagram to be crosscut before execution by traceEntry(String)and after execution by traceExit(String).

5.4.2.2.4 Integration specification

Theme supports the specification of two types of integration - override and merge. Due to the symmetric model that theme supports, integration relationships can be specified between different theme types. Integration specification is denoted by an arc and can be specified at two levels.

Firstly, integration can be specified at a theme level. This indicates that all elements at the arc endpoints must be composed in accordance to any integration rules specified by that arc. An example of a basic integration rule is match<name>. This indicates that overlapping is recognized by name and specifies that where names match, all elements of matching name should converge during composition.

Secondly, integration can be specified between elements within theme encapsulations. To illustrate this case, we follow on from our previous example. Where classes in themes overlap but do not have the same name, a lower level of specification is required to specify that these classes should converge. Override and merge integration can be specified at both levels.

Override integration is a specification of structural crosscutting. It is used to specify that an overlap between theme elements is to be resolved by replacing the elements in existing themes with the theme elements that are overriding. Override integration is between theme elements and can specify the replacement of elements across themes. As such, override integration can be considered to be a specification of structural crosscutting. Override integration is denoted by a one way arc.

Merge integration is a specification of structural crosscutting. Merge integration is used to specify that an overlap between theme elements is to be resolved by adding the parts of overlapping elements together. The result of a merge is the sum of the parts of those elements specified to be merged. This result is an altered structure in that merged structures are compositions of structures. A merge specification can be merge elements between many themes. Therefore, it can be considered to be a specification for structural crosscutting. Merge integration is denoted by a two way arc.

Merge integration between base and aspect themes are specifications for crosscutting behaviour. As noted in section 4.3.1, aspect themes are parameterized templates. Arguments that match the parameters are specified by bind<> expressions associated with merge integration specification. A bind specification is illustrated in Figure 5-14. In a bind relationship join points are specified or a criterion for join point selection is specified. In either case the join points are the behavioural points to be crosscut by the behaviour encapsulated in the aspect theme. In Figure 5-14, the bind specification uses wildcards (denoted by *) to indicate that all methods in all classes encapsulated in theme S1 are to be crosscut by the trace theme.

5.4.2.2.5 Composition semantics

Detailed composition semantics are defined for Theme/UML in [158]. These semantics are represented more generally in [159].

5.4.2.3 AOD Process

The Theme process is described as a three phase process [160] – analysis, design and composition. In the analysis phase, themes are identified and characterised. In the design phase the identified and characterised themes are specified in technical design models. Finally, in the composition phase, the composition of themes is specified. The entire process is illustrated in Figure 5-16.

The Analysis part with Theme/DOC has been discussed earlier (section 3.5.2.1). In short, Theme/Doc supports the characterisation of two theme types, aspect and base themes. Base theme views present the designer with the requirements for the base themes that are needed to produce a design model that fits the base theme requirements. Aspect theme views present the designer with the requirements for the aspect theme. Aspect theme views also present the themes that are crosscut by the aspect.

The design phase, shown as the second step in the Theme process in Figure 5-16, takes each theme and produces separate and possibly overlapping models. The requirements related to the theme view are used to construct theme designs for each theme. Each theme is constructed as a class diagram in a separate theme construct. A theme construct is a specialised package. Aspect themes require an extra sequence diagram to be produced which indicates where and how the aspect theme is to crosscut the behaviour of another theme.

Once the themes are captured in separate and possibly overlapping design models the composition phase begins. This is presented as the final process at the bottom of Figure 5-16. In this phase, the integration or composition of themes is specified. Classes in Theme/UML model domain concepts. Overlapping design models are those where the same domain concept is represented in more than one theme as a class. An overlap occurs when the requirements for different themes partially describe a domain concept. Here the domain concept is fully described across all themes in which the domain concept expressed.

During the composition process, overlapping domain concepts must be composed. The composition of these domain concepts is specified during the composition phase. Aspect themes are expressed as parameterised template packages in Theme/UML. In order to specify the integration or composition of aspect themes with other themes, it is necessary to specify the join point's selection criterion. This indicates where the aspect theme crosscuts the other themes. The specified selection criterion facilitates the identification of the points in the related themes that are to be crosscut by the aspect theme. These points are used as arguments to the parameterised template package. Once the arguments are bound to the template, a composition can be realised.

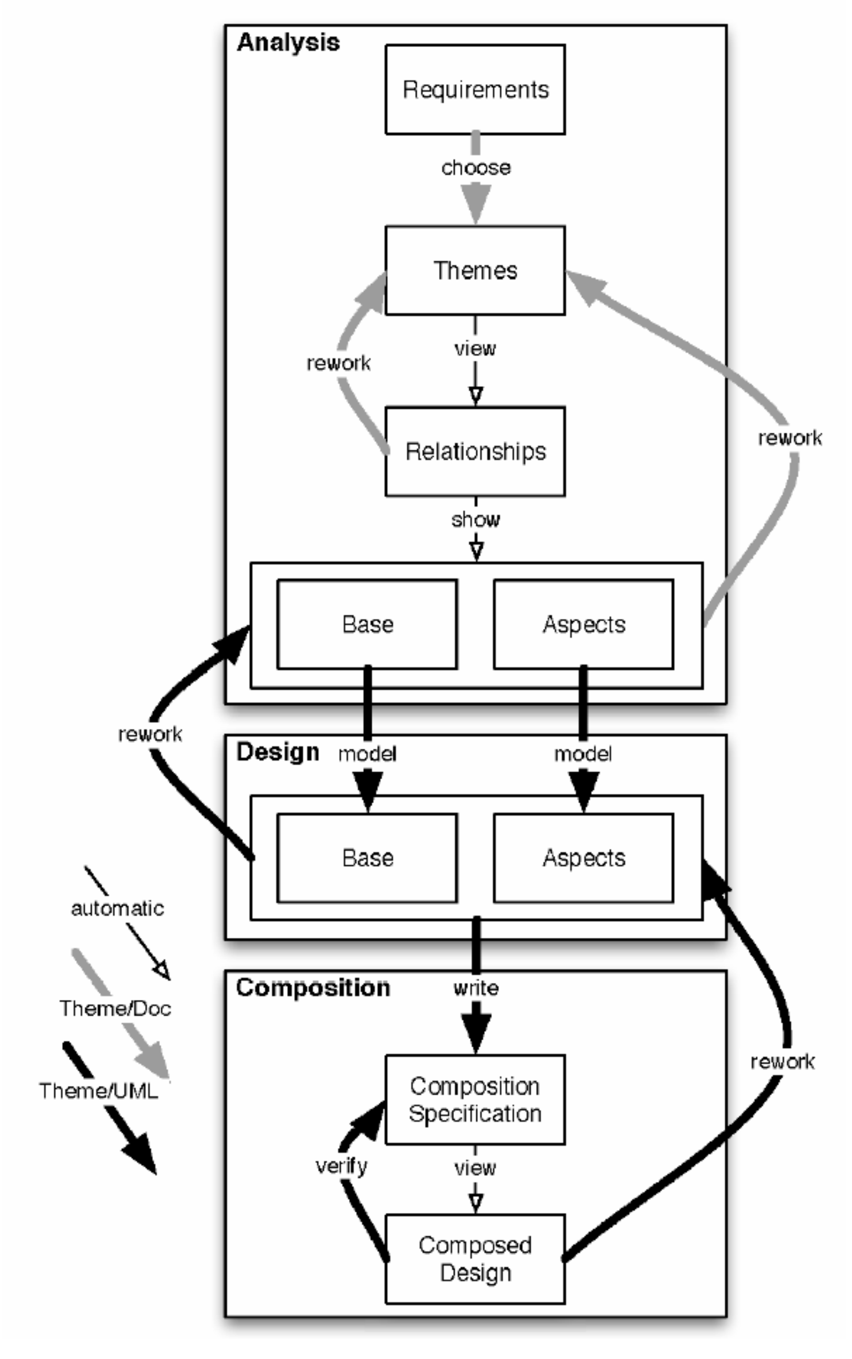


Figure 5-16 Theme process [160]

The result of overlapping concept composition is a model in which all domain concepts are fully represented in a model. The result of composing aspect themes is the generation of models where the join points identified by the specified selection criteria are used as arguments to the parameters in the aspect theme templates.

Although not described in Figure 5-16, Theme also provides guidelines for a process for the implementation of theme designs on the AspectJ, HyperJ, AspectWerkz and CME implementation platforms [160-162].

The theme process and its relationship with existing design processes are discussed in [160]. The theme process is said to usable in a waterfall, iterative or agile context.

5.4.3 State charts and UML Profile (SUP)

5.4.3.1 SUP Method:

The SUP approach supports a process for AO analysis and design based on state charts, complemented with an AOD Language based on a UML profile.

5.4.3.2 SUP AOD Artefacts

5.4.3.2.1 Language

To support this process an AOD Language specified as a UML profile is proposed in [142, 163]. This profile models base and aspect structure in class diagrams. Behaviour is modelled in state chart, use case, state machine and collaboration diagrams. A set of stereotypes that extend the semantics of existing UML elements are used across these design diagrams.

5.4.3.2.2 Specification of Aspects

Aspects are structural specified in class diagrams. Aspects are specified as using an <<aspect>> stereotype. Aspects can be specified as being synchronous or asynchronous through the inclusion or exclusion of an <synchronous> tag. Synchronous aspects alter the control flow and asynchronous aspects do not.

Aspects are also specified in state chart diagrams. In this sense the aspect is specified as a set of states that are connected through a series of events.

5.4.3.2.3 Specification of Crosscutting

The specification of behavioural crosscutting is achieved through the use of state charts [164]. Here crosscutting behaviour is modelled as an event that triggers a state transition.

5.4.3.2.4 Integration specification

Specification of behavioural crosscutting is done in two ways. Firstly, in class diagrams, integration is specified through an association type that indicates crosscutting. This association is denoted by the <<crosscut>> stereotype. The <<crosscut>> association supports symmetric integration specification.

Secondly, in state charts, integration is specified through linking events across state chart diagrams. Aspects, as stated in Sections 5.3.1 and 5.3.2, can be specified in state chart diagrams and crosscutting behaviour is modelled as linking events (by name) between state chart diagrams. Integration is specified as these linked events. Linking indicates that when the base event is triggered, the aspect event that is linked to that event should also execute.

5.4.3.2.5 Composition semantics

This approach provides relatively informal composition semantics. As discussed, integration is specified through linking events across state diagrams. Composition occurs when events are broadcast from one state diagram to other state diagrams. Any events that are linked to the event broadcasted are activated in the state diagrams to which the event is broadcast. These events can then cause an alteration of the states of many modules in the overall system.

5.4.4 SUP AOD Process

The SUP process is based on the use of state charts. The process is described and illustrated as a series of steps in [165] and [164]. The process is used to extract aspect concerns from an OO concern description by identifying the crosscutting state transitions that exist in the OO model. Once identified, the aspects and base concerns can be specified separately during design.

Analysis begins with requirements for a base concern which are tangled with descriptions of aspect concerns. These requirements are used to create a state chart diagram. This state chart diagram represents the state transitions for the concern. Through analysis of the state chart diagram the aspect concerns and are identified and the relationship with the main concern is captured.

Once base and aspects concerns are identified, class diagrams are used to describe the static structure of the base and aspect concerns. Aspects and base concerns are specified as classes. Operations and attributes for the classes can be derived from the state chart diagram developed during analysis.

Crosscutting relationships between base and aspect behaviours are specified in state chart diagrams. Each state diagram is specified separately and it has its own states. The events that trigger transitions between states are shared by aspects and base state chart diagrams. As such the events that are shared between base and aspect classes can be considered as join points.

When events occur the event is described as being broadcast. As such, the aspects are notified of the event and the invocation of crosscutting behaviour is modelled as a change in state of the aspect class.

In [166] the use of state charts is linked with Feature Driven Development [136]. State charts in this instance are used to specify features. A process of incremental feature development is supported though feature composition being specified in state charts.

5.4.5 Aspect-Oriented Architecture Modelling (AAM)

5.4.5.1 Aspect-Oriented Architecture Modelling Method

Aspect-oriented Architecture Modelling (AAM) is an approach that focuses on specifying concerns at middle to high design levels. This approach is based on role based meta-modelling and uses UML as a basis for an AOD language [167].

5.4.5.2 Aspect-Oriented Architecture Modelling Artefacts

5.4.5.2.1 AOD Language

AAM as an AOD language supports two types of aspects - context-free and context specific aspects. Context free aspects are reusable aspects that are expressed at a high level. Context specific aspects are aspects that are instances of context free aspects that can be specified for use (or reuse) in specific design models.

5.4.5.2.2 Specification of Aspects

High level aspects are specified as parameterized template package diagrams. Parameters are explicitly demarked using the "|" symbol in the diagrams. Parameters are described in this way to avoid listing the parameters in the package header. Listing the parameters in the package header is described as unwieldy when there are many parameters.

The diagrams encapsulated in an aspect parameterized template package describe the aspects in terms of structure and behaviour. Structurally, aspects are modelled in class diagrams. Within the class diagram, template classes, their template members and their inter-template relationships are used to describe the structural properties and constraints that a middle level aspect must contain to realise the high level aspect.

Aspect behaviour is modelled in collaboration diagrams. Within the collaboration diagram the interactions between object templates are described. The sequenced interactions between the object templates are described in terms of the messages sent between object templates.

5.4.5.2.3 Specification of Crosscutting

A high level aspect can be completely modelled in terms of an abstract primary model. Design elements external to the primary solution can be used to describe relationships between the primary model and elements introduced through the aspect model.

A high level aspect introduces relationships and constraints to the primary design model, by binding the context of the primary model to an abstract primary model described in the high level aspect.

The context that realizes the context specific aspects is identified through bindings between the context free aspects and the primary model. Only certain primary model context can be bound to high level aspects. Before context can be bound to high level aspects, the context must be validated.

Pre and post conditions specified in OCL defined conditions that determine the validity of a context binding. Each template has associated with it a set of OCL context constraints. These constraints ensure that any context bindings that are used to ensure that middle level aspects are valid.

Structural crosscutting is specified in context specific aspects. Structural crosscutting is specified in the class diagrams that are generated from the template class diagrams described in the high level aspect.

Behavioural crosscutting is specified in context specific aspects. Behavioural crosscutting is specified in the collaboration diagrams that are generated from the template collaboration diagrams described in the high level aspect.

5.4.5.2.4 Integration specification

Integration of middle level aspects and a primary model is specified through composition directives [168]. Composition directives specify how primary and aspect models are to be merged. Model merging is based on model overlap between aspect and primary models and is described in Section 4.3. Composition directives specify the addition of elements to primary models, removal of elements from primary models, renaming elements in primary models, changing references within primary models, overriding elements in primary models and a specification to order aspect composition with the primary model.

Composition directives are divided into two categories - high level and low level. Low level composition directives are used to specify the composition of a single aspect model and a primary model. High level composition directives specify the composition of many aspect models with the primary model. Composition directives constraints are specified in OCL.

5.4.5.2.5 Composition semantics

Composition semantics associated with AAM are best outlined in [168]. The composition semantics associated with AAM are similar to Theme/UML. The semantics differ based on the levels of separation supported by each.

Composed models are both structural and behavioural in representation. Structurally, composition is specified in a class diagram in which the aspect models and primary model are composed. Behavioural composition is specified in a collaboration diagram.

5.4.5.3 AOD Process

AAM is not explicitly discussed in terms of an AOD process. Figure 5-17 illustrates the overall AAM model. From this model we can infer or assume some of the primary steps that one may follow when using the AAM approach.

In analysis the primary model is identified and the high level aspects that are needed to realize the concerns that crosscut the primary model are identified. If there are no high level aspects that capture a particular crosscutting concern, a middle level aspect is required to realize that concern.

In the design phase the primary model is expressed in a middle level design. At this point, the designer can assess the high level aspect designs available to see if context required to realize the high level aspects in a middle level design, is available in the primary model.

When the relevant high level aspects and context is available, then bindings for these high level aspects to the application is specified. The composition of high level aspects and application context are middle level (or context sensitive) aspects.

Middle level (or context sensitive) aspects are then composed with the primary model.

Composition directives specify the order and manner in which a middle level aspect and primary models are to be composed. The result of applying the context sensitive aspects to the primary model, as specified in the composition directives is a fully composed OO model expressed in UML.

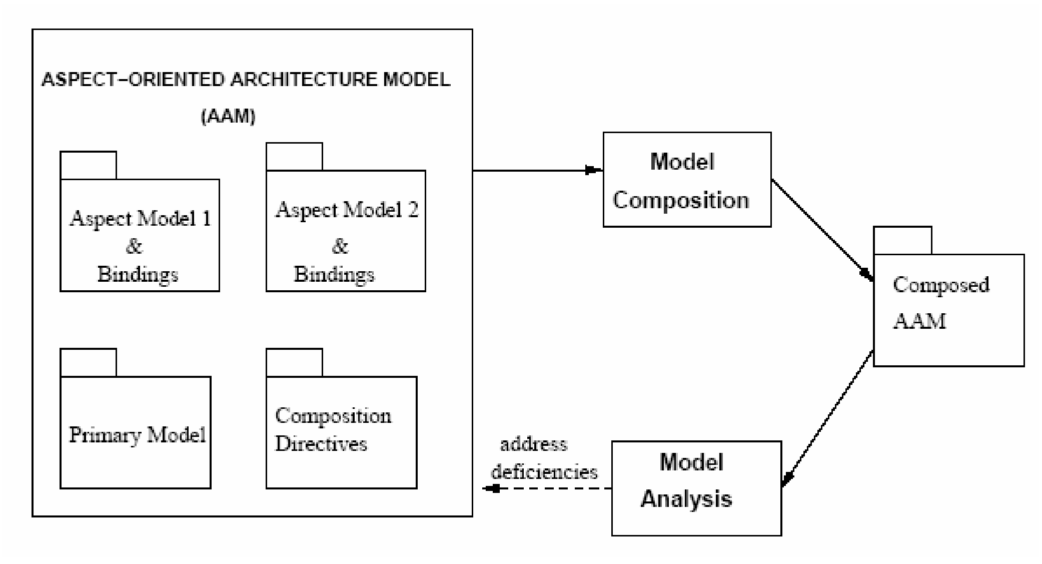


Figure 5-17 Components of the AOM Approach [167]

5.4.6 CoCompose

5.4.6.1 CoCompose Method

CoCompose is a (non-UML) design language that supports the representation of high level reusable aspects as features [169]. CoCompose introduces a feature construct in the CoCompose AOD language. A feature is a high level aspect that crosscuts application boundaries.

5.4.6.1.1 AOD language

CoCompose is a graphical design language that can be used to support AOD. CoCompose supports executable designs by design language elements being semantically well-defined.

Design Algebra is a technique for determining and selecting concrete language constructs for implementing concepts. This is basis for the CoCompose design language to programming language translation.

5.4.6.1.2 Specification of aspects

Reusable aspects are specified as features in CoCompose. Features are abstract constructs that describe design patterns. Features are semantically well defined. A feature is a composition of concepts and roles. A role is a template concept. Concepts are the basic language construct in CoCompose and are used to model domain concepts.

Concepts can have implementation strategies associated with them. Where concepts have no implementation strategies associated with them, a suitable implementation strategy may be provided by the feature in which the concept is expressed.

5.4.6.1.3 Specification of crosscutting

Roles are templates that must be fulfilled to realise features as concrete entities. Roles are specifications of crosscutting. They have associated constraints that restrict and validate the concepts that are used to realise or match the role.

5.4.6.1.4 Integration specification

Solution patterns are CoCompose constructs that specify the integration of a features roles and concepts.

5.4.6.1.5 Specification of Composition

Once a solution pattern integrates the concepts, a full concept model is the result. As implementation strategies are linked with concepts and features, a composition can be treated as an executable system. Once composition is complete the design model can be translated into an implementation.

5.4.6.1.6 Composition semantics

Composition semantics are not well defined for CoCompose.

5.4.6.2 CoCompose Process

CoCompose is aimed at automating the design to implementation process. Automating code generation is achieved through defining implementation strategies for semantically well-defined design elements. As such, this approach is focused on automating the transformation from high level design models to platform specific implementations.

5.4.7 UML For Aspects (UFA)

5.4.7.1 UML For Aspects Method

UFA, or UML For Aspects, is an AOD approach based on the Aspectual Collaborations Model (ACM) [170]. UFA is an extension of UML.

5.4.7.2 UML For Aspects Artefacts

5.4.7.2.1 AOD Language

UFA is an extension to the UML which supports symmetric AOD.

5.4.7.2.2 Specification of aspects

Reusable or high level aspects are modelled independent of the context in which they will be applied. An aspect is specified as an abstract package. In UFA, UML packages are used for encapsulating parts of a system that contribute to a complex behaviour. To extend the semantics provided by UML packages defining top–level properties (attributes and methods) of packages is allow. This encourages using a package as a façade. Graphically, this is achieved by adding one or two compartments to the box representing a package, one for attributes and the other for methods.

As illustrated in Figure 5-18, packages are tagged as abstract. Aspects are defined abstractly to support reuse. The implication is that reusable packages are incomplete. Packages are reused through specialization and parameterization. Direct parameterization is not used because explicit parameters are only convenient when used in a small number and with little structure. Aspect packages contain class diagrams that represent the detailed design of the complex behaviour being modelled. Abstract classes and methods defined with aspect packages are roles that need to be fulfilled to realize the aspect in the context of an application. Role classes need to be specialized and adapted to a specific application.

5.4.7.2.3 Specification of crosscutting

Specialization allows the aspect be specialized to crosscut a specific application. The adaptation relationship indicates that application context is bound to the aspect specialization to adapt the abstract aspect to the specific application context. As illustrated in Figure 5-18, an adaptation relationship plus a specialization relationship specify a <<connector>>. A <<connector>> is middle level aspect design that is specific to the application.

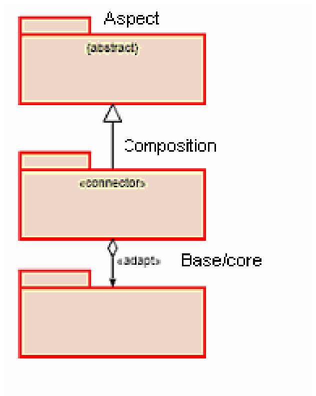


Figure 5-18 Package level Composition

An aspects role classes and methods are bound to application context within the <<connector>>. This type of binding is called a “callout”. A callout is represented by an arrow moving from the role method to a method in the core class. Fulfilling the method can be achieved in two ways.

The first way is to delegate from the role method to an existing method in the design module class. Delegation is used where a role and context are semantically similar but syntactically differ The second way is to have a direct syntactic and semantic match between aspect and application elements affected, in which role is directly specialized by the application design elements A callout can be considered as a specification of structural crosscutting.

Weaving methods into the core model is another way of binding application and role classes. This type of binding is called a “callin”. A callin is represented in by an arrow moving from the core method to a method in the role class. Callin bindings specified as before, after, or replace. A callout can be considered as a specification of behavioural crosscutting.

Callin and callout bindings are visual represented on the connector package. Callin and callout bindings are specified in OCL.

5.4.7.2.4 Integration specification

At the package level the connector package represents integration specification. The connector package is a specialisation of an abstract package that contains a crosscutting specification and adapts a package representing core functionality. At a class level, the binding of roles to core classes specifies an integration of classes and roles. At method level the callin and callout relationships between methods specify integration.

5.4.7.2.5 Composition semantics

The composition semantics for UFA are not well defined.

5.4.8 Uml eXchange Format (UXF)

5.4.8.1.1 Uml eXchange Format Method:

UFX focuses on support AOD information exchange between UML case tools [171]. Specifically, the UXF approach introduces an XML based language called UXF/a. UXF/a as an exchange language is based on an AOD extension of the UML meta-model.

5.4.8.2 Uml eXchange Format Artefacts

5.4.8.2.1 UXF AOD Language

In Figure 5-19 a segment of the UML meta-model extension shows that an aspect is defined as a construct derived from the classifier element. A classifier is an abstract UML meta-class that describes structural and behavioural features. Aspects are described as having attributes, operations and relationships. Aspects relationships include generalisation, association, and dependency. Aspects are in effect represented as <<aspect>> stereotyped class like constructs.

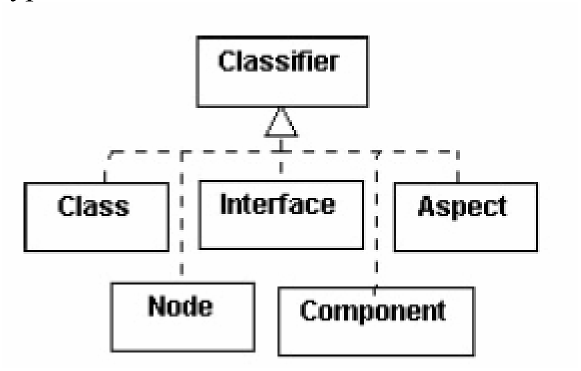


Figure 5-19 Aspect as a classifier [171]

The aspect-class relationship is defined as an extension of the dependency relationship defined in the UML meta-model. As illustrated in Figure 5-20, the meta-model defines structure and relationship for describing the composition of aspects and classes.

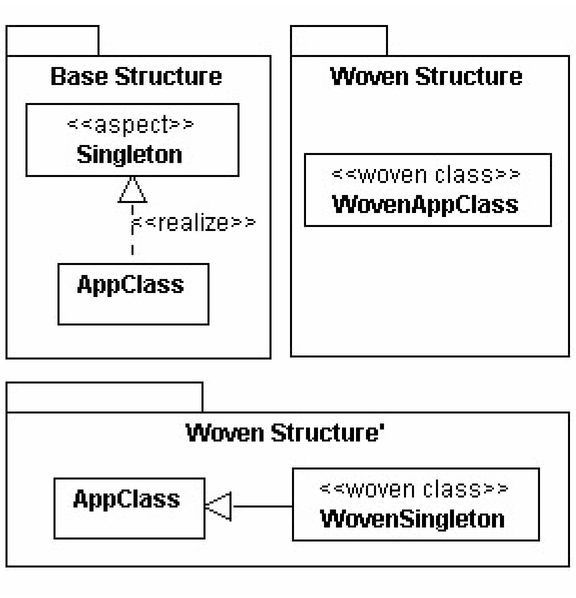


Figure 5-20 Composition [171]

5.4.8.2.2 Specification of aspects

An aspect is specified by the presence of a classifier of the <<aspect>> stereotype.

5.4.8.2.3 Integration specification

Integration specification is a stereotyped relationship between classes and aspects. The stereotype describes and is dependant on the type of integration being specified.

5.4.8.2.4 Composition semantics

Composition semantics are not well defined for UXF.

5.4.8.3 Uml eXchange Format AOD Process

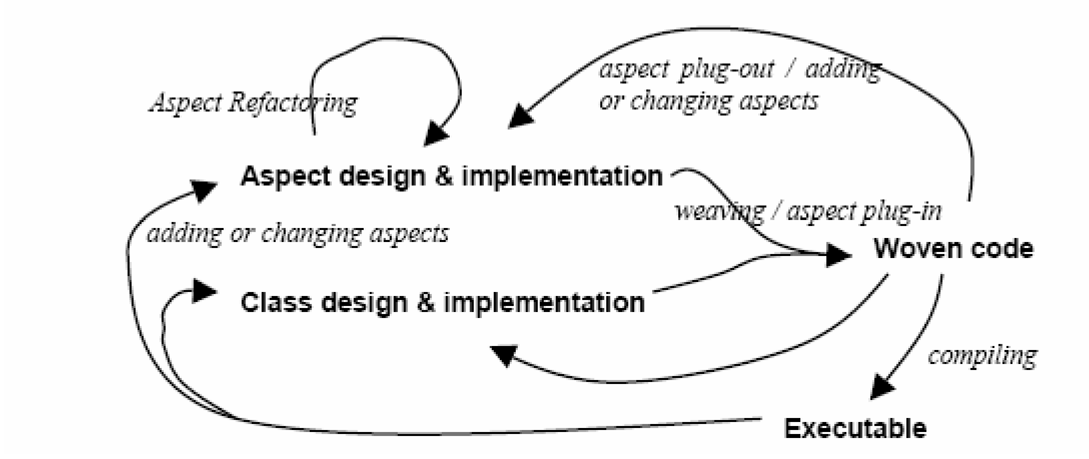

Aspect Refactoring
aspect plug-out / adding or changing aspects
Aspect design & implementation
weaving / aspect plug-in
adding or changing aspects
Woven code
Class design & implementation
compiling
Executable

Figure 5-21 A typical process of aspect-oriented development [171]

In [171] UXF support for a development process that is based on stepwise refinement is described. In this process, illustrated in Figure 5-21, aspects are designed and implemented separately from classes. Classes and aspects are interwoven and then executed. After weaving or execution we assume there is some verification, followed by appropriate refactoring of the design.

5.4.9 Architectural Views of Aspects

5.4.9.1 Architectural Views of Aspects Method

Architectural Views of Aspects (AVA) builds on superposition-position based design [172]. AVA recognises two problems with current AOD approaches. The first problem is that there is no generic way to reason about the influence on one aspect of some other aspect of the same system. The second is that there is not much provision for the incremental design of aspects [173].

These problems arise because aspects are treated as being independent of each other. Overlap between aspects exists when more than one aspect implements one concern. This overlap means that, although aspects are separate entities, they are related and are not completely independent.

Sub-aspects are aspects that are not completely orthogonal. Sub-aspects can be composed to form a composite aspect, which represents a single concern. Due to a possible dependency between sub-aspects, the order in which sub-aspects are applied to other concerns, or are composed may be important. Identifying sub-aspects and their interdependencies is important to ensure that sub-aspects work in unison.

This approach introduces a Concern Architecture Viewtype (CAV). A CAV is defined to group sub-aspect designs that represent one concern. Within a CAV, dependency relationships are specified between multiple sub-aspect designs, that when composed represent a concern. These relationships are specifications to ensure that sub-aspect compositions reflect the correct behaviour of the concern.

Concern architectures are one view type. It is noted that other view types that capture other perspectives can complement the concern architecture view type.

5.4.9.2 Architectural Views of Aspects Artefacts

5.4.9.2.1 AVA AOD Language

A concern architecture is a concern model described in a software architecture view type. A view type specifies the types of elements and relationships which can be used to describe software architecture. A concern view type specifies the types of elements and relationships that can be used to describe a concern. An aspect describes a crosscutting concern. An aspect view type specifies the types of elements and relationships that can be used to describe a crosscutting concern. An aspect view types described in [173] is the basis for an AOD language. This AOD language addresses both high level and middle level design modelling.

5.4.9.2.2 Specification of aspects

Crosscutting concerns are described as being modelled as one or more aspects. As such, an aspect can be a composition of sub-aspects, which are aspects in their own right. Aspects themselves are described as being a composition of *required* and *provided* parts. The required part of an aspect describes the context that corresponds to join points, at which provided parts are introduced. Provided parts are then specifications of crosscutting. Aspects are described as having an interface. A subset of the provided parts can be designated as *hidden*. The provided parts of an aspect that are not hidden are exposed through this interface and hidden parts are not. As such, aspects can be modelled at different levels. High level aspects completely represent one crosscutting concern. Low level aspects represent parts of potentially many crosscutting concerns. Lower level aspects offer their non-hidden provided parts, to satisfy the required parts of other aspects.

A high level aspect is depicted as an encircled collection of low level or sub-aspects. The high level aspect is dependant on sub-aspects. Dependency is illustrated in Figure 5-23 as a <<concern>> stereotyped dependency relationship. Aspects are specified in packages which are stereotyped with the <<aspect>> stereotype.

Aspects packages contain structural and behavioural diagrams. These can be used to specify aspect behaviour. Class diagrams are used to model aspect structure. Sequence and state diagrams are used to model aspect behaviour.

5.4.9.2.3 Specification of crosscutting

As illustrated in Figure 5-22, structural crosscutting is specified in class diagrams. The classes in these class diagrams are tagged as required or provided. As mentioned in Section 11.3.1, the provided parts are then specifications of crosscutting. As such, the classes tagged as provided are specifications of structural crosscutting.

As illustrated in Figure 5-22, behavioural crosscutting can be specified in state diagrams. The states and events in state diagrams are tagged as required or provided. As mentioned above, the provided parts are then specifications of crosscutting. As such, states and events tagged as provided are specifications of behavioural crosscutting. Other behavioural diagrams, such as sequence diagrams, can be used in a manner similar to state charts, to describe aspect behaviour.

5.4.9.2.4 Integration specification

Integration is specified at two levels in this approach. The first is at the package level, where one aspect package is specified as an integration of sub-aspect packages. In this case, a composite aspect is specified as ordered dependencies on sub-aspects which are required by the composite. This composite aspect package can then crosscut the other concerns. Another way to view an aspect, is as a group of sub-aspects that individually crosscut other concerns. In this case, the order of that application is defined for the application of the sub-aspects to the concerns that they collectively crosscut. In both cases, the order in which sub-aspects are composed or applied is of importance due to their inter-dependence. To specify an order for aspect integration an "aspect1/aspect2/aspect3" notation is used. Figure 5-23 provides an example, where this notation can be used to describe aspect dependency. In the example, there are three aspects, where C is used by both O and S (so there is an arrow from each of them to C). The integration of C into O and S is specified as a stereotyped <<crosscut>> relationship. There are two possible composition orders of these sub-aspects O and S, these include O/S/C and S/O/C ("O using S using C" and "S using O using C", respectively).

Integration is specified at two levels in this approach. The first is at the package level, where one aspect package is specified as an integration of sub-aspect packages. In this case, a composite aspect is specified as ordered dependencies on sub-aspects which are required by the composite. This composite aspect package can then crosscut the other

concerns. Another way to view an aspect, is as a group of sub-aspects that individually crosscut other concerns. In this case, the order of that application is defined for the application of the sub-aspects to the concerns that they collectively crosscut. In both cases, the order in which sub-aspects are composed or applied is of importance due to their inter-dependence. To specify an order for aspect integration an "aspect1/aspect2/aspect3" notation is used. Figure 5-23 provides an example, where this notation can be used to describe aspect dependency. In the example, there are three aspects, O and S are sub-aspects and C is a composite aspect. The integration of O and S into C is specified as a stereotyped <<crosscut>> relationship. There are two possible composition orders of these sub-aspects O and S, these include O/S/C and S/O/C.

The second level at which integration can be specified, is between the design elements within the aspect package. As mentioned in Section 11.3.1, class and state chart diagrams are used to specify the aspects structure and behaviour. As such, integration is specified between the elements specified in these diagrams. Integration at this level is specified as composition binding relationships. The types of composition binding relationships supported by this approach include - regular binding, replacement and unification. These relationships specify the elements to be integrated and how they are to be integrated.

Figure 5-22 illustrates several examples of composition relationships. Binding is characterised as the required part of one element being bound to required part of another. On the left side of Figure 5-22, the provided part of class C, in <<aspect>> A, is bound to the required part of class D, in <<aspect>> B, through the <<bind>> relationship. A bind relationship is a high level specification of integration, where the providing class augments the provided class. Returning to our example, class C augments class D. Augmentation amounts to the merger of overlapping aspects. (Overlapping concerns are discussed in Section 4.)

Replacement is characterised by the provided part of one element replacing the provided part of another element. The middle diagram in Figure 5-22 illustrates an example of a replacement integration specification. Elements of the same name appear in <<aspect>> A and B. These elements represent the states that aspects A and B can be in. Transition between states is triggered by the execution of the provided parts of the aspects. Execution is depicted through events. In the example, the provided part of <<aspect>>, named E_event, is replaced by the provided parts of <<aspect>> B, named F_event and G_event. This replacement is legal, as E_event allows a transition from state S_state to T_state, F_event facilitates a transition from state S_state to U_state and G_event causes a transition from state U_state to T_state. As such, the state transitions in <<aspect>> A are augmented through the <<replace>> relationship imposed by <<aspect>> B.

Unification occurs when distinctly named elements are renamed such that they become identical. This can be considered a merger of the provided parts of aspects A and B. An example of a unification relationship is illustrated on the right side of Figure 5-22.

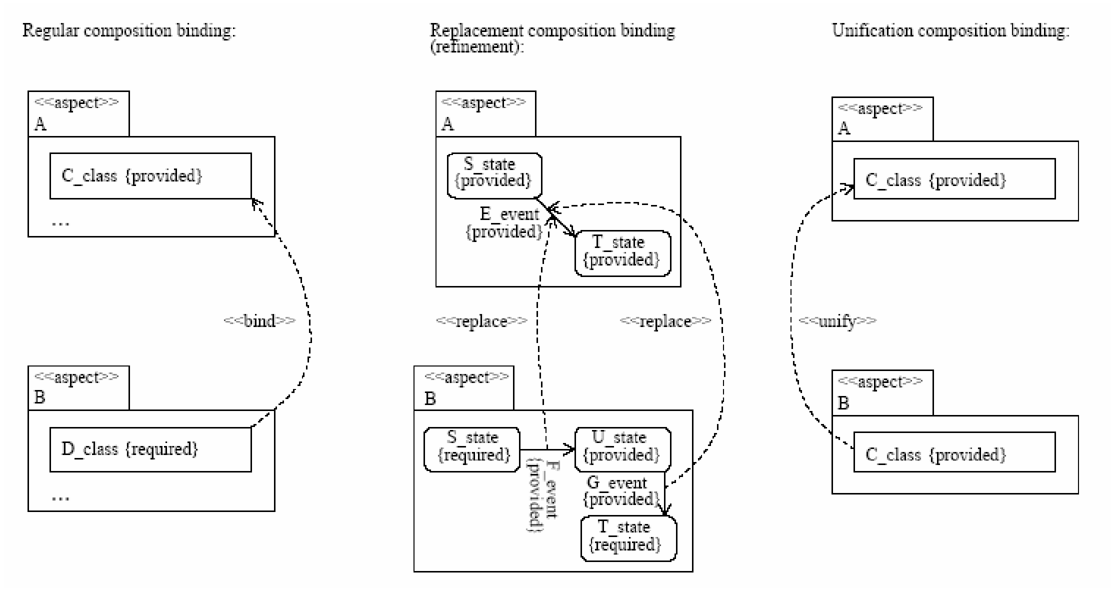


Figure 5-22 Superimposition Binding Mechanisms [173]

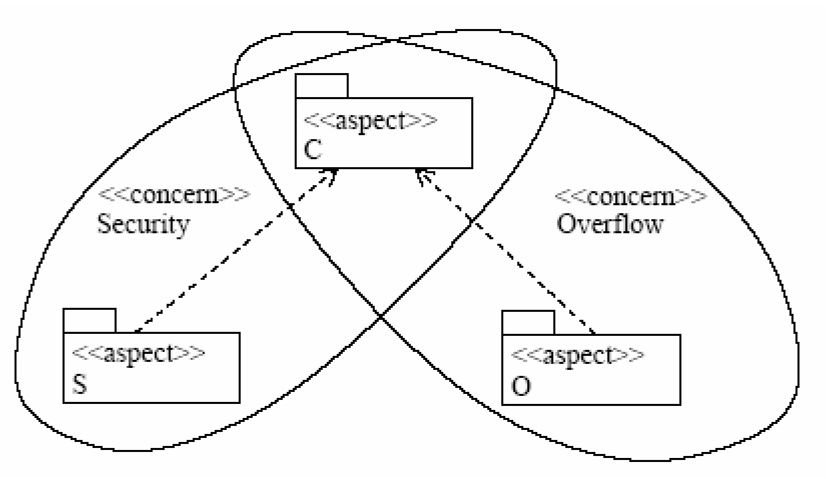


Figure 5-23 Concern architecture [173]

5.4.9.2.5 Composition semantics

Composition semantics are required at two levels in this approach. As aspects can be compositions of aspects, the semantics of sub-aspect composition to form aspects is required at a package level. Within aspect packages, class diagrams represent structure, state and possibly other diagrams that represent the behaviour associated with that structure. The integration of these, more detailed diagrams, is specified through bind, replace and unify relationships. Aspects are described as well formed "if and only if the corresponding artefact in language L obtained by ignoring the required/provided/hidden tags are well formed". Composition is well formed when a "well formed aspect B can augment another well formed aspect A with given bindings if and only if the resulting composite aspect B/A is well formed". Well-formedness rules are specified in OCL.

5.4.10 Aspect Modelling Language (AML)

5.4.10.1 Aspect Modelling Language Method

The Aspect Modelling Language (AML) supports middle-to-low level design [174-176]. AML is aimed at supporting aspect reuse, by following the UFA approach, and described Section 9. AML is a UML based notation that allows support for forward engineered of designs into implementation.

5.4.10.2 Aspect Modelling Language Artefacts

5.4.10.2.1 Aspect Modelling Language AOD Language

AML is based on a subset of the AspectJ language constructs and the design notation described in UFA.

5.4.10.2.2 Specification of aspects

Like UFA, AML considers aspects to be higher level constructs than classes. Aspects are specified as packages of the <<aspect>> stereotype.

5.4.10.2.3 Specification of crosscutting

Aspect packages encapsulate crosscutting structure and behaviour. The connector package contains the crosscutting structure and behaviour specification. The association between the connector and aspect packages is shown by using a <<uses>> stereotyped dependency relationship. As illustrated in Figure 5-24, the aspect uses the connector to specify the crosscutting structural and behavioural features, encapsulated by the aspect.

In the connector package, introductions are specified in an <<introduction>> stereotyped classifier. This contains intertype declarations specified in the AspectJ style. These intertype declarations reference structural elements declared in the aspect.

In the connector package, advice is specified in a classifier stereotyped as <<advice>>. Within this classifier, advice is specified in the AspectJ style. The type of advice is indicated through a <<before>>, <<after>> or <<around>> stereotype. Advice declarations reference elements declared in the aspect.

5.4.10.2.4 Integration specification

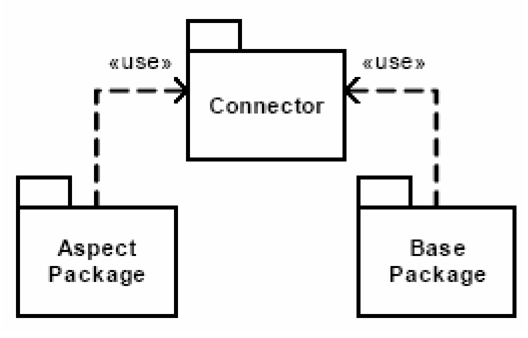


Figure 5-24 AML Aspect, connector, base [176]

Integration is specified in the connector package. A connector package encapsulates a pointcut classifier, an introduction classifier and an advice classifier. These constructs represent the AspectJ programming constructs. As illustrated in Figure 5-24, a connector package represents a crosscutting relationship between an aspect and potentially many base packages.

The introduction, pointcut and advice constructs are represented as stereotyped classifiers. A <<pointcut>> class contains the pointcuts related to an aspect. Pointcuts themselves are represented as method like signatures of the form <pointcut name [!] execution point>. The [!] symbol indicates that the method is a pointcut. The name is a reference to the point of execution identified. At the class level, this specifies the execution point identified in the pointcut as a point of integration.

5.4.10.2.5 Composition semantics

Composition semantics follow the AspectJ implementation of AOP.

5.4.11 Aspects At Design Time (ADT)

5.4.11.1 Aspects At Design Time Method

ADT, or Aspects at Design Time, is an approach to AOD that proposes extensions to UML [177]. This approach was an early proposal for AOD. ADT approach focuses on representing synchronisation as separate concern.

5.4.11.2 Aspects At Design Time Artefacts

5.4.11.2.1 ADT AOD Language

This language is focused on enabling the separation of a synchronisation concern.

5.4.11.2.2 Specification of aspects

A class of the <<synchronised>> stereotype represents a synchronisation concern. A <<synchronised>> class models synchronisation for an application. Synchronisation behaviour is specified a state diagram that corresponds to the class.

5.4.11.2.3 Specification of crosscutting

Crosscutting elements are specified in the <<synchronised>> class and in a corresponding state diagram that models that class.

5.4.11.2.4 Integration specification

This integration is specified through a <<synchronise>> association between the class, that encapsulates the synchronisation concern, and those classes which it crosscuts. Actions are specified for this <<synchronise>> association. These actions reference events described in the state diagram. This reference implies that when a call that is defined on a class is crosscut, the synchronisation concern is invoked and the crosscutting behaviour expressed in the state diagram is executed.

5.4.11.2.5 Composition semantics

Composition semantics for ADT are not well defined.

5.4.12 Aspect Oriented Component Engineering (AOCE)

5.4.12.1 Aspect Oriented Component Engineering Method

Aspect Oriented Component Engineering (AOCE) is a software development methodology that describes aspects as services, provided to and required by components [53] AOCE supports the identification, description and reasoning about components and aspects. In AOCE components and aspects are composed at run-time. AOCE as an approach to AOD facilitates modelling of dynamic AO systems.

5.4.12.2 Aspect Oriented Component Engineering Artefacts

5.4.12.2.1 AOCE AOD Language

AOCE supports a design language based on component based dynamic AOP. This approach supports design of applications that target a component based dynamic AOP implementation platform.

5.4.12.2.2 Specification of aspects

Aspects are specified as components in AOCE. Components are related through the services they provide and require. Components can also be specified at a higher level, in a detailed design model. The definition of a component as an aspect is separate to the definition of the component. Components are defined in a component model in AOCE. AOCE extends the component model with an aspect model. The component/aspect design model is representation of the underlying implementation framework. The framework is extended to create application specific designs.

The aspect model supports separate aspectual characterisation of a component. Components are extensions of a type that links the component with an AspectManager. The AspectManager, and associated types, are representations of the AOCE component/aspect framework. The AspectManager has associated with it AspectDetails. AspectDetails encapsulate the crosscutting and integration specification for components. In AOCE there are different specialisations of AspectDetail that capture specific features for encapsulating the crosscutting and integration specification for components.

Aspects are also specified in component diagrams. The aspect-component runtime interactions can be specified in collaboration diagrams.

5.4.12.2.3 Specification of crosscutting

The crosscutting part or provided part of a component is specified at runtime through information encapsulated in instances of AspectDetails.

In collaboration diagrams, the provided part of a component is specified as a crosscutting interaction with components.

5.4.12.2.4 Integration specification

The join point part or required part of a component is specified at runtime through information encapsulated in instances of AspectDetails.

In collaboration diagrams, integrate is specified as interactions between the required and provided parts of components.

In detailed collaboration diagrams, specify how the aspect will integrate with the components that the aspect crosscuts through specifying interactions.

5.4.12.2.5 Composition semantics

Composition in AOCE occurs at run-time. Composition semantics are specific to the AOCE implementation platform.

5.4.12.3 AOCE Process

Components are first identified and characterised using conventional OO analysis. The resulting components are analysed, in terms of the services provided by, and used by each component. Related services are grouped, and categorised as being required (used) by, or provided by the component.

Functional and non-functional features of the system are identified. Service relationships are distinguished by their association with a feature. For each group of services, the services required by the component are associated with the components that provide those services. The services provided by the group are associated with components that require those services.

As all components are assessed in a similar fashion, the result is a matrix in which all services can be commonly grouped. This matrix provides views in which component features can be categorised into conceptual groupings. Provided services can be expressed once and related to all the places that it is required. This conceptual grouping is considered to be an "aggregate aspect".

Aspect details describe the aggregate aspect. Aspect details describe aspects and their relationship with other aspects. Aspects and aspect details can be described textually in a vocabulary skewed toward component based software development. Alternatively they can be graphically represented as an extension to OOD diagrams (see [53]).

Once identified, aspect details are refined into software component aspects that categorise design level component services. Refinement begins by associating aspects, identified during analysis, with design models. Design in AOCE is not implementation neutral. Design is focused on implementation specification.

AOCE designs can be implemented on any component framework. AOCE uses an AO extension of component platforms to implement AOCE designs. AO platforms enable the decoupling of components at implementation that have been decoupled at design when following the AOCE methodology.

Where run-time composition is possible coupling only occurs at run-time. Run-time composition can be illustrated in design by collaboration diagrams however the relevant elements of the run-time composition environment that are used in composition must be expressed in this diagram.

5.4.12.4 UML for AOSD Method

5.4.12.4.1 UML for AOSD AOD Language

A UML notation for AOSD [178] (UML4AO) is a proposed UML extension to support the design of AO programs. The significant extensions are the introduction of groups, pointcut relations and aspect classes. Groups provide means for the classification of heterogeneous and distributed entities. Pointcut relations allow the developer to define crosscuts within the program. Aspect classes implement the extension of the program based on the join points identified by the pointcut relations.

This approach supports design focused on the JAC framework. The primary goals of the JAC framework are supporting dynamicity and distribution. The JAC framework is also a general purpose AOP environment based on AspectJ concepts. The UML extension defined provides a design view of the JAC facilities and language.

5.4.12.4.2 Specification of aspects

Aspects are specified as aspect-classes. An aspect-class is a classifier that contains aspect-methods. These constructs are similar to regular classes and methods, but differ semantically, due to their crosscutting nature. An aspect class is specified through an <<aspect>> stereotype.

5.4.12.4.3 Specification of crosscutting

UML4AO supports behavioural and structural crosscutting. To specify crosscutting several stereotypes are provided. <<before>>, <<after>> and <<around>> stereotypes are associated with aspect-methods specify behavioural crosscutting. Structural crosscutting is specified by the <<replace>> and <<role>> stereotypes. The <<role>> stereotype indicates that the element it is associated with is to be introduced into the base classes the aspect crosscuts. The <<replace>> stereotype indicates that the element it is associated with is to replace elements in the base classes the aspect crosscuts.

5.4.12.4.4 Integration specification

A pointcut relation allows the designer to indicate where aspect-methods crosscut base-methods. The crosscutting association between the two is indicated at the classifier level, where the aspect-class crosscuts the base-class. This relationship is defined by a <<pointcut>> stereotyped association.

A group, in UML4AO, represents a group of design elements, which may be heterogeneous, that are identified by a pointcut. The UML does not naturally support grouping of heterogeneous objects. The group notation facilitates reasoning about group-typed sets of objects. The group notation allows a designer to map a concern space to a set of objects. Grouping can be done based on the location (host/container) of an object instance. A <<group>> stereotype is used to identify groups.

5.4.12.4.5 Composition semantics

As UML4AO is based on the JAC framework, the composition semantics associated with this language are similar to the composition semantics of JAC.

5.4.12.5 TranSAT Process

TranSAT can be viewed as a process in two distinct ways. The first view is TranSAT supports the entire software development lifecycle as an iterative evolutionary process. In this view, concerns are merged together until a system is complete. This assumes that the core model is a composite concern that grows at each evolution of the architecture. Concerns are completely developed independently of the core and then integrated with the core. During iterations the core grows and evolves.

The second view is that TranSAT applies to the maintenance phase of a software development lifecycle. In this view, development is finished and a core architecture and system exists. Evolution through expanding the architecture with new technical concerns could be considered to be maintenance.

Taking either view, a set of steps is outlined for using TranSAT. When abstracted away from TranSAT, these steps can be considered a high level process description for concern integration.

The process assumes an existing architecture or core model. A concern is independently specified. Once a concern is specified, the rules to integrate the concern with a generic component framework are conceived. These integration rules are developed a generic architecture model. As such, this provides rules that allow the concern to be integrated with any architecture that matches the generic architecture model. When a target architecture requires that concern, the meta-model is matched against the target framework. The points that the integration rules identify, in the architecture meta-model, are reified in the target architecture. Pointcuts can then be devised to facilitate the integration.

5.4.13 Component Views

5.4.14 Component Views Method

View components are introduced in [179] to promote reuse of functional concerns across component based systems. View components are an extension of components models, based on views.

A concern view over components is a decomposition of the components based on the components relationship with a concern. Each view represents a concern and each view contains the components and component relationships that are part of a concern. A system space is a set of systems that share common concerns.

In [180] views abstraction and reuse across concern boundaries is explained. In abstracting views the components and inter-component relationships related to a concern are abstracted. Component functionality can also be captured in this abstraction. Concerns can be then be reused in other applications. An abstract view component can be reused by mapping the abstract elements of the view component to the application elements. The functionality preserved in the abstract components can be reused in other application contexts.

View components define roles that have an associated reusable behaviour and a reusable relationship between roles. Concerns can be reused by filling the roles expressed in a view component with applications entities.

To represent these concepts in a design language, a meta-model for view components is proposed as an extension to UML 1.4. This meta-model (Figure 5-25) introduces the concept of a connection between an abstract view components element and a base (or application) element.

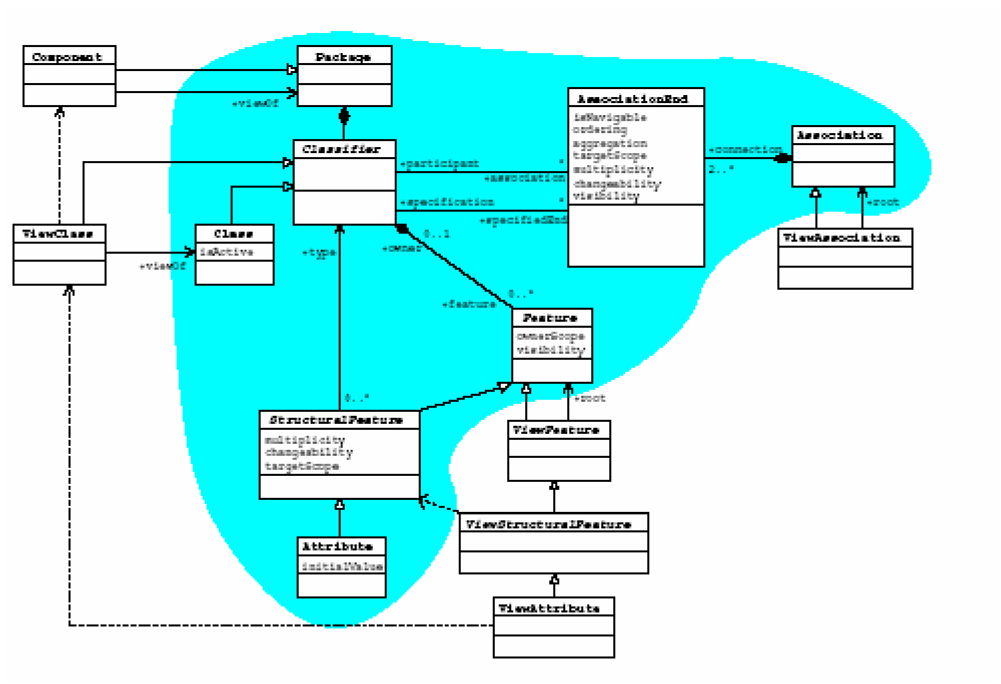


Figure 5-25 View Components Meta-Model [179]

The meta-model defines an abstracted component that represents crosscutting concerns as ViewClasses. A ViewClass is a specialization of the Classifier UML meta-class, and is defined as a class that can be applied to a collection of classes. A Component is a specialization of the package meta-class and represents an abstracted concern. Components are made up of ViewClasses and Classes. The structural and behavioural features of ViewClasses are specified as ViewStructuralFeature and ViewBehavioralFeature elements. A ViewAssociation is a specialization of the Association UML meta-class. An instance of this type makes it possible to specify an association between ViewClasses.

Constraints for the meta-model are expressed in OCL. Design constrains are used to check the coherence at the modelling phase. Association constraints are used to check that the system obtained by the assembly of packages is coherent.

5.4.14.1 Component Views Artefacts

5.4.14.1.1 Specification of aspects

Aspects are specified as components. Components contain ViewClasses.

5.4.14.1.2 Specification of crosscutting

View classes contain ViewBehavioralFeatures and ViewBehavioralFeature. ViewBehavioralFeatures specify crosscutting behaviour and ViewBehavioralFeature specify crosscutting structure.

5.4.14.1.3 Integration specification

Integration specified by mapping abstract components design to concrete application design. As such, the abstract elements that are aggregated within the component are realised with a concrete application structure.

5.4.14.1.4 Composition semantics

Composition semantics are not well defined for the View components approach.

5.4.15 Activity Diagrams

5.4.15.1 Activity Diagrams Method

UML 2.0 introduces a new version of activity diagrams [130]. This differs from UML 1.X versions of activity diagrams as it separates activity from state diagrams and is based on Petri net semantics.

A Petri net is an approach for modelling systems characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic. It consists of places, transitions, and arcs that connect them. Input arcs connect places with transitions, while output arcs start at a transition and end at a place. There are different types of arcs, e.g. inhibitor arcs. Places can contain tokens; the current state of the modelled system is given by the number of tokens in each place. Transitions are active components. They model activities which can occur, thus changing the state of the system. Transitions are only allowed to fire if they are enabled, which means that all the preconditions for the activity must be fulfilled. When the transition fires, it removes tokens from its input places and adds some at all of its output places. The number of tokens removed/added depends on the cardinality of each arc. As such there are multiple token flows for each activity.

Due to the redefinition of activity semantics with Petri nets, modelling of activities is more natural as activities can have multiple flows at any one time. This is different to UML 1.X where multiple tokens were used only during the behaviour occurring between a fork and join.

Activity diagrams define activities and the relationships between activities. Activities represent behaviour in software. The relationships between activities represent the flow between activities.

Activities can be composed hierarchically. An activity defined with an activity diagram can refer to another activity diagram that describes that activity in greater detail. We refer activities that are composed into higher level activities, as sub-activities.

Activity diagrams by their nature are aimed at modelling complex systems. As such, activity diagrams can be complex and graphically tangled. The only form of decomposition available in this instance is the breaking of activities into hierarchies of sub activities (or vertical decomposition). This form of decomposition does not facilitate the modelling of crosscutting concerns. In [147] and [146] horizontal decomposition of activity diagrams is introduced.

Three types of activity nodes are defined for activity diagrams that are to be composed horizontally - interface nodes, activity nodes and subtraction nodes. Addition nodes are activity nodes that are added to an activity diagram during horizontal composition. Subtraction nodes are activity nodes that are removed from an activity diagram during horizontal composition. An interface node represents the point at which the horizontally decomposed activity diagram can be integrated into an orthogonal activity diagram.

A UML profile specifies the semantics of the three activity node types. Stereotypes are introduced for the subtraction activity node and the interface activity node. No stereotype is introduced for addition activity nodes as these are considered the default semantic for nodes that appear in an aspect activity diagram.

In [146] a compelling, yet simple example is provided, and is seen here in Figure 5-26. This illustrates the horizontal composition (Process Order form) of an activity diagram (Order handling) and an initially unrelated aspect activity diagram (invoice handling).

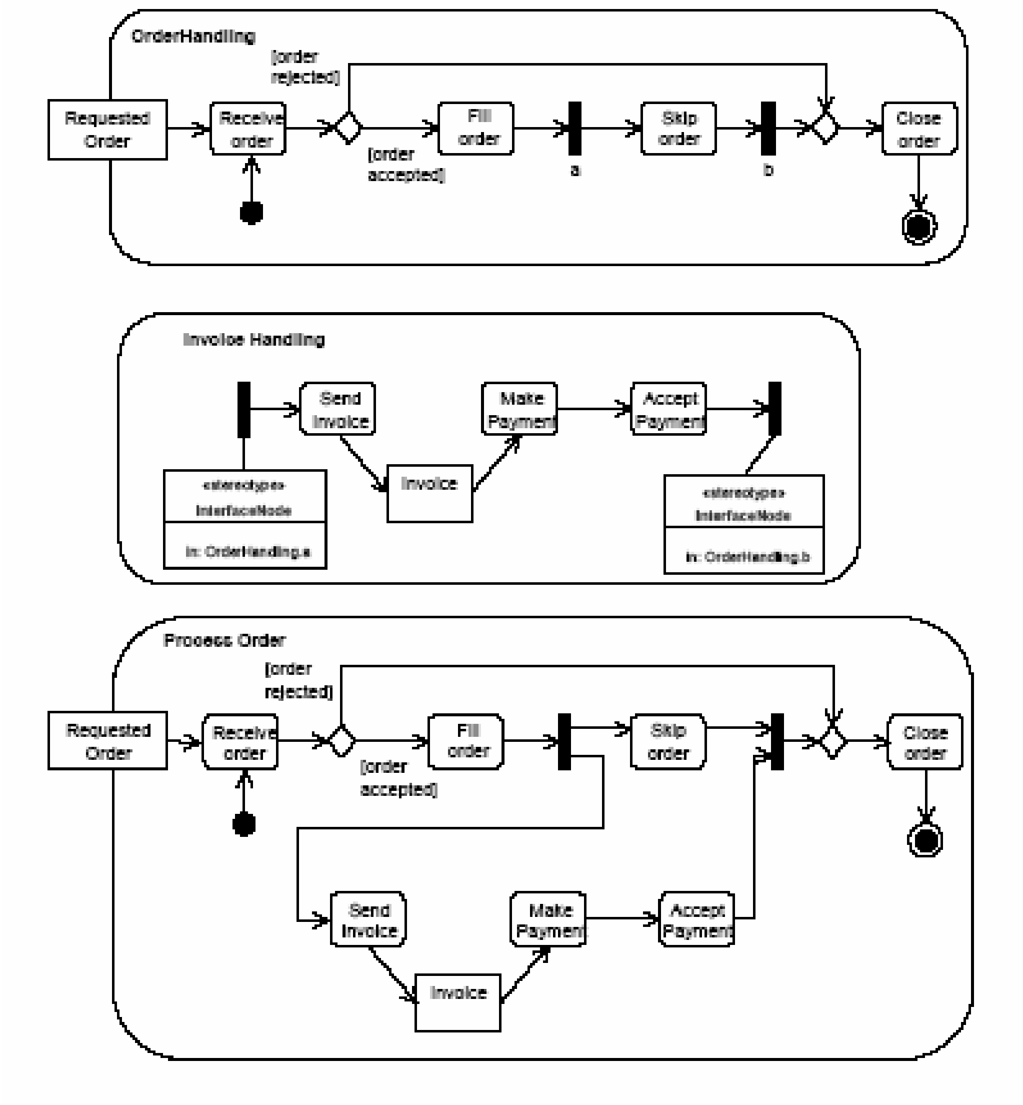


Figure 5-26 Activity Diagram Composition [146]

If we look at the Order Handling activity diagram, we notice that the forks in the diagram are named *a* and *b*, respectively. Notice also, that these names are referenced in the Invoice Handling stereotyped interface nodes that identify the forks present in the Invoice Handling diagram as points of horizontal integration for the Invoice Handling diagram which here is an aspect activity diagram. Fork *a* and *b* can be considered to be join points in this case. The composition of these diagrams is presented in the Process Order Diagram. We can see that the activities that are invoked in between the forks in the aspect activity diagram have now been added to the initial activity diagram, at the forks identified as *a* and *b*. Further examples are available in [146].

5.4.15.2 Activity Diagrams Artefacts

5.4.15.2.1 Activity Diagrams AOD Language

The ability to capture crosscutting behaviour in activity diagrams in UML 2.0 is clearly demonstrated here.

5.4.15.2.2 Specification of aspects

Aspects are specified as specialised activity diagrams that use the activity addition profile.

5.4.15.2.3 Specification of crosscutting

Addition and subtraction activity nodes in the activity diagram are specifications of crosscutting.

5.4.15.2.4 Integration specification

Integration is specified in interface activity nodes. These nodes reference nodes in target activity diagrams at which integration is to take place.

5.4.15.2.5 Composition Semantics

Basic composition semantics for activity diagram composition are specified in [147] and [146].

5.4.16 CAM/DAOP

5.4.16.1 CAM/DAOP Method

CAM/DAOP is a component and aspect based approach that supports the separation of concerns through the software development lifecycle, from design to implementation. This approach defines the CAM (Component-Aspect Model) [181] model, the DAOP-ADL language [122] and the DAOP (Dynamic Aspect-Oriented Platform) platform. The DAOP platform is out of the scope of this document.

CAM is a component and aspect based model that defines an extension of UML to specify how to design the structure of an application in terms of components, aspects and the composition among them. The information provided with CAM during the design is then expressed in terms of an XML document using the DAOP-ADL language. This information is directly consulted and used by the DAOP platform at runtime. The different levels of modelling aspects in CAM/DAOP are related among them by using MDA.

5.4.16.2 CAM/DAOP Artefacts

5.4.16.2.1 CAM/DAOP AOD Language

CAM is an AOD language based on components and aspects. CAM supports symmetric concern decomposition where the core functionality of the system is modelled as any number of different components and the concerns that crosscut this core functionality are modelled as aspects.

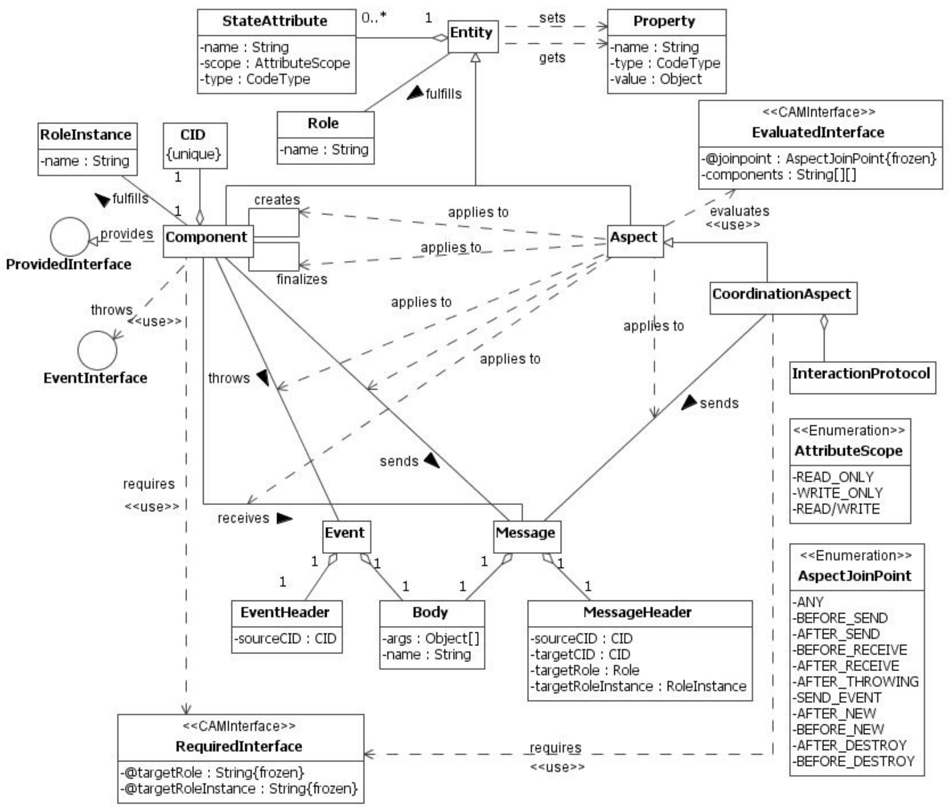


Figure 5-27 [64] The Component and Aspect Model

Figure 5-27 shows an UML diagram with the basic entities of the CAM model and the relationships that can be established among them. This diagram is considered as the metamodel of CAM (part of the UML Profile for CAM), so the entity names are UML stereotypes for modelling applications in terms of CAM.

5.4.16.2.2 Specification of aspects, integration and crosscutting

In CAM those concerns that are modelled as components are identified with the stereotype <<Component>>, while those concerns that are modelled as aspects are identified with the stereotype <<Aspect>>.

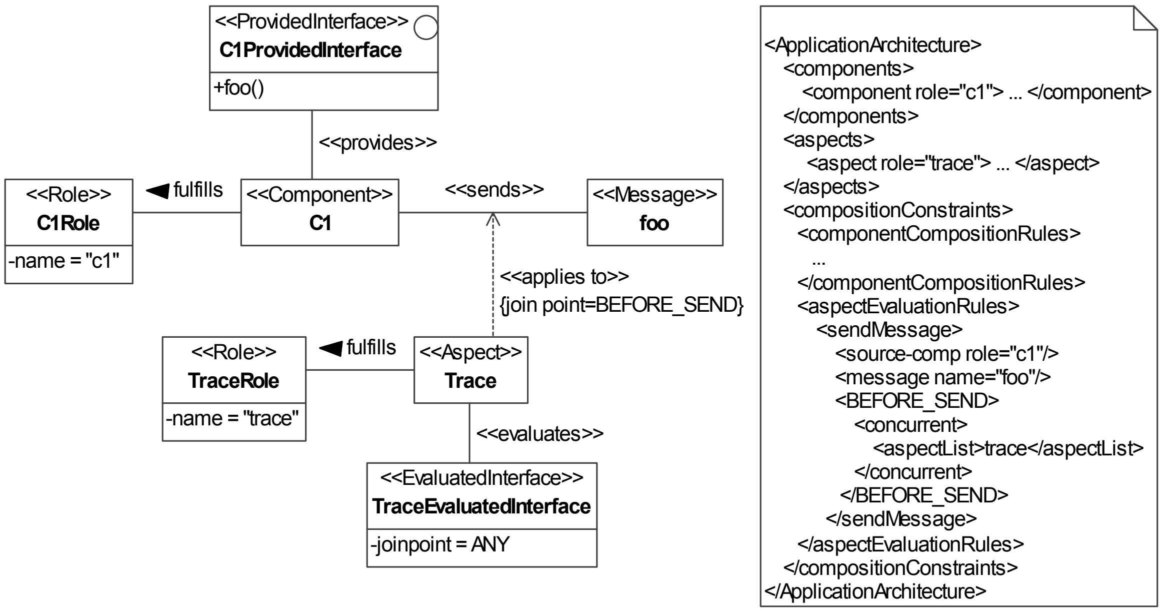
As shown in Figure 5-27, in the CAM model aspects are treated as a "special" kind of component and therefore, both share some common features. Both have a set of attributes stereotyped with <<StateAttributes>> that represent their public state, both are identified by a unique role name (<<Role>>) to identify the specific functionality played by the component or the aspect, and both make use of the concept of property (<<Property>>). For instance, properties are used to decouple non-orthogonal aspects, which do not need to directly interact among them to resolve their dependences. Instead, they indirectly interact by sharing properties with the same name and type.

In CAM both components and aspects can act as units of composition with contractually specified interfaces and explicit context dependencies. CAM describes the *provided* and *required* interface of components to describe both the services the component provides to the environment and also those services it requires in its

[64] Source of Figure: [181] Figure 1 (extended).

interaction with other components (see Figure 5-27). For aspects CAM describes the *aspect evaluated* interface, which includes information about the join points an aspect is able to intercept and evaluate (see Figure 5-27). The internal behaviour of components and aspects can be described using standard UML models. CAM does not constraint or gives specific guidelines about it.

Figure 5-28 shows an example of the partial design of an application in CAM. It shows one component and an aspect and the relationship among them. As shown in Figure 5-28, the information provided with CAM is expressed in XML. This is done using the XML-based DAOP-ADL language. This diagram describes that when a component with role name "c1" sends the "foo" message the aspect with role name "trace" is applied before sending the message (join point BEFORE_SEND).



```
<ApplicationArchitecture>
  <components>
    <component role="c1"> ... </component>
  </components>
  <aspects>
    <aspect role="trace"> ... </aspect>
  </aspects>
  <compositionConstraints>
   <componentCompositionRules>
    ...
   </componentCompositionRules>
   <aspectEvaluationRules>
     <sendMessage>
       <source-comp role="c1"/>
       <message name="foo"/>
       <BEFORE_SEND>
         <concurrent>
           <aspectList>trace</aspectList>
         </concurrent>
       </BEFORE_SEND>
     </sendMessage>
   </aspectEvaluationRules>
  </compositionConstraints>
</ApplicationArchitecture>
```

Figure 5-28 An example of CAM Component and Aspect Integration

5.4.16.2.3 Specification of crosscutting

In CAM the specification of crosscutting between components and aspects is expressed in terms of the "*applies to*" relationships shown in Figure 5-27. CAM was previously classified as an approach with a low-middle concern abstraction. This is reflected in the way in which CAM specifies crosscutting where the join point intercepted by the aspect is specified.

CAM considers that aspects are composed with black-box components and therefore this model intentionally avoid the definition of join points that intercept the internal behaviour of components, and only have access to components through its public interface. Therefore, CAM defines that aspects can crosscut the behaviour of components before and after (incoming and outgoing) messages and events, and also before and after the creation and destruction of component instances (see Figure 5-27). The "applies to" relationships in Figure 5-27 can be tagged with the specific join points in which an aspect should be applied. This can be seen in the example in Figure 5-28 in order to indicate that the aspect with role name "trace" crosscuts the component with role name "c1" before the component sends the "foo" message (tagged value *joinpoint=BEFORE_SEND*).

For non-orthogonal aspects, the crosscutting between aspects is expressed in terms of CAM Properties. Non-orthogonal aspects indirectly interact to resolve their dependencies by sharing properties with the same name and type. For example, let us suppose an authentication aspect that requests the user's name and password in order to authenticate him or her; and a filter aspect that filters the messages received by a component according to the user's preferences. In order to get the user preferences the filter aspect needs to know the user's name and this name is previously obtained by the authentication aspect, creating dependences among them. In order to solve this dependency, the authentication and the filter aspects may share a property with name "username". The value of the property will be established by the authentication aspect when the user is authenticated and consulted by the filter aspect when needed.

5.4.16.2.4 Integration Specification

CAM integrates the specification of all the components and aspects that set up an application by describing the software architecture of the application using the DAOP-ADL language [122]. This language describes all the components and the aspects that model the application and the integration among them.

In addition to describe the components and aspects through their role names, interfaces, etc, the DAOP-ADL language describes: (1) how to compose components among them and, (2) how to compose aspects with components. The prior are expressed in the *componentCompositionRules* section of the XML document (see right side of Figure 5-28) and are described in terms of the role name of source and target components. The later are expressed in the *aspectEvaluationRules* section of the same document (see again right side of Figure 5-28) and are expressed in terms of the role names of the source and the target components, the role name of aspects and the join points in which aspects crosscuts the source and/or target components. In Figure 5-28, the *sendMessage* eval rule in the XML document in the right side translates the information expressed in UML with the CAM model.

5.4.16.2.5 Composition Semantics

The design of a CAM application provides information about the composition among components, the relationship among aspects and the composition between components and aspects.

The composition among components is expressed as part of the *componentCompositionRules* section of the DAOP-ADL language in terms of the source and target role names (see right side of Figure 5-28). Additionally, the information provided during the description of their provided and required interfaces determines how they can be composed among them.

The relationship among aspects is expressed using a UML activity diagram, where each aspect is represented as an *activity* as shown in Figure 5-29. With this diagram CAM indicates if several aspects being evaluated in the same join point are evaluated in sequence or concurrently. Usually, this decision will depend on the orthogonality of aspects. Orthogonal aspects may be evaluated both in sequence and concurrently, while non-orthogonal aspects (sharing common CAM properties) should be evaluated in sequence. This information is also expressed with the DAOP-ADL language in the *aspectEvaluationRules* section (see right side of Figure 5-28).

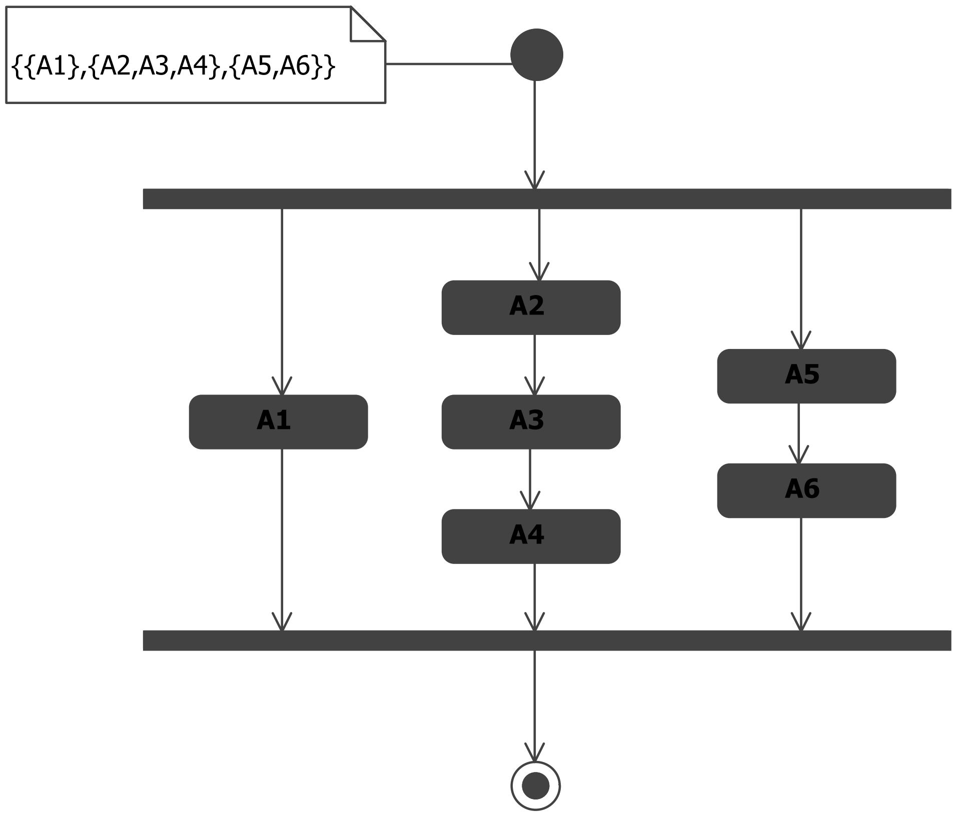


Figure 5-29 [65] Activity Diagram showing Aspect Relationship in CAM

Finally, the composition among components and aspects is expressed in CAM by means of the "*applies to"* dependency relationships shown in Figure 5-27. For each of these relationships the CAM model has a tagged value indicating the join point that the aspect is intercepting and another tagged value indicating the criticality of the aspect. CAM aspects are classified as either *critical* aspects or *non-critical* aspects, depending on how important the result of the evaluation of that aspect is for the behaviour of the final application. Once again all this information is expressed with the DAOP-ADL language in the *aspectEvaluationRules* section.

5.4.16.2.6 CAM/DAOP & MDA Process

CAM/DAOP uses MDA to enable the implementation of a methodology for creating systems based on the CAM/DAOP approach. This process allows the refinement of the system from an OO representation of the design to a DAOP middleware specific implementation as is shown in Figure 5-30.

In [182] an example of this AOD process is described in MDA. MDA is used to formalise the models and transformation between models that are created during design. By using MDA this process begins with a *computational model* of an application. A computational model is a basic model of the application entities. This model is then marked and transformed into a CAM model based on the CAM profile. The CAM model describes the computational model in terms of components, aspects and the relationships among them. The CAM model (the PIM) is then marked and transformed into a DAOP model (the PSM). The DAOP model represents an implementation independent meta-model for the DAOP approach. It also includes the

[65] Source of Figure: [182] Figure 3 (extended)

description of the architecture of the application with DAOP-ADL, which is automatically generated from the CAM model. The DAOP in turn is marked for transformation to a specific implementation platform.

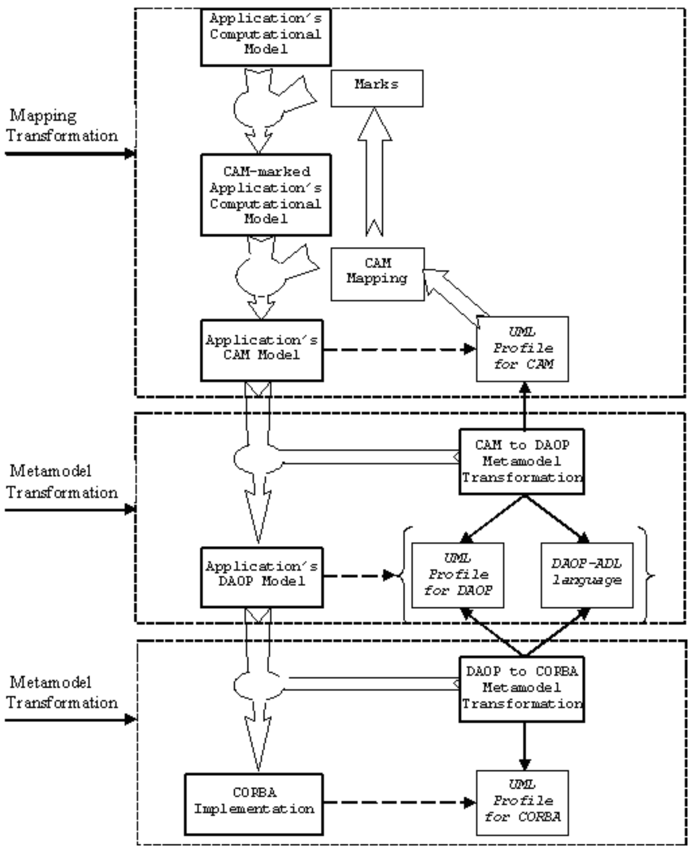


Figure 5-30 The MDA Stack of Models for CAM/DAOP

5.4.17 Implementation Driven Aspect Modelling (IDAM) Approach and AOP-to-UML Approach

5.4.17.1 Implementation Driven Aspect Modelling and AOP-to-UML Approach Methods

Implementation driven aspect modelling (IDAM) [183] proposes an integration of aspect oriented programming and model driven development. To support this integration, an approach to model aspects is required. Contemporary asymmetric approaches to aspect modelling associate aspects with core models. This is achieved through parameterization, and binding or directly relating aspect and core models. The former uses bind statements which can be complex to interpret, while the latter is said to introduce graphical tangling. A new visualization called dynamic aspect diagrams is suggested to overcome these problems.

Dynamic aspect diagrams (DAD) are user responsive diagrams, that model AspectJ constructs (aspects, inter-type declarations, pointcuts and advice). Associations between aspects and core elements are visualized dynamically, that is to say, the user controls what aspect-to-base associations are visible in a diagram. DAD's are generated from AspectJ code.

AOP-to-UML is another code driven modelling approach described in [184]. In this approach UML extension mechanisms are used to create models based on AspectJ code.

5.4.17.2 Implementation Driven Aspect Modelling Artefacts

5.4.17.2.1 Implementation Driven Aspect Modelling AOD Language

DAD's are loosely based on UML class diagrams. UML constructs are modified rather than extended.

5.4.17.2.2 Specification of crosscutting

The DAD's specification of crosscutting is based on AspectJ. The crosscutting structure is integrated directly into the core model and is not specified in DAD's. Crosscutting behaviour is specified in an advice design element. Crosscutting behaviour and structure is encapsulated in an aspect. These design elements are representations of the AspectJ language constructs and share the semantic value of these constructs.

5.4.17.2.3 Specification of aspects

An aspect classifier is introduced to represent aspects. It is similar to the class classifier except that it encapsulates AspectJ member types and relationships. An aspect supports two relationships not specified in the UML. The "advises" relationship (Figure 5-31) associates an advice body with a point in the cores structure. A call relationship indicates a method call. It is represented by a specialization of the UML association relationship.

5.4.17.2.4 Integration specification

Integration specification is illustrated by an icon. This icon, seen in Figure 5-31, is an arrow enclosed in a circle and indicates the presence of a crosscutting relationship. The "advises" relationship specifies that integration of an aspect and core classes. The icons indicate where integration will occur.

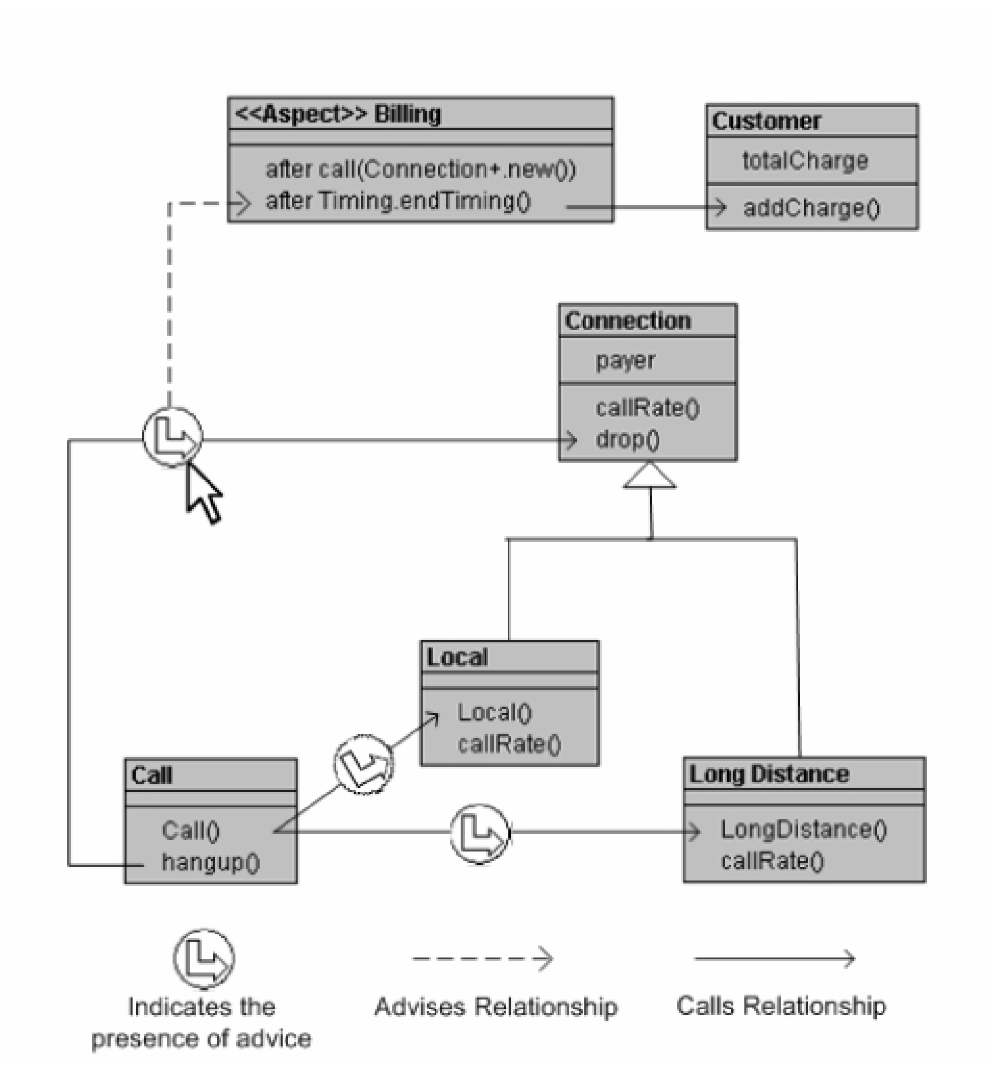


Figure 5-31 Dynamic aspect diagram [183]

5.4.17.2.5 Composition semantics

Structural composition semantics are the same as AspectJ. Crosscutting behaviour is executed after (as opposed to before) an integration point in the base program's execution. This is indicated with the arrow icon.

5.4.17.3 AOP-to-UML Artefacts

5.4.17.3.1 AOP-to-UML AOD Language

The AOP-to-UML design approach is based on the observation that logging behaviour, encapsulated as advice in AspectJ, crosscuts classes based on method signatures, expressed in the pointcuts associated with that advice. A method invocation is represented as a message in UML collaboration diagrams. A message connects two types, the type that invokes the method and the type that exposes the method. A connection point is a point of association between a type and a message. A message between two collaborating types has two connection points. An incoming connection point is where the message originates and an outgoing connection point is where the message is handled. An aspect can introduce extra behaviour in between incoming and outgoing connection points. This increases the number of connection points that can be related to a message. A call pointcut identifies an incoming connection point. Advice associated with the pointcut is introduced at the incoming connection point and introduces second outgoing connection point related to the coming connection point.

This approach is based on extending the UML to introduce representations of connection points as well as introducing design elements to represent AspectJ constructs.

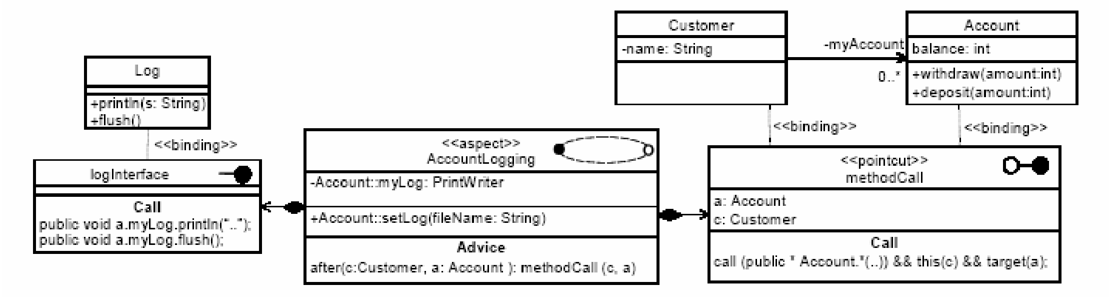


Figure 5-32 Logging Aspect [184]

5.4.17.3.2 Specification of crosscutting

Figure 5-32 illustrates the design of a crosscutting logging behaviour. The logging behaviour is encapsulated in the Log class. The Log class has its own structure and also behaviour. Crosscutting behaviour is represented in methods signatures as in normal UML. An outgoing connection point specifies that these methods are crosscutting. Connection points are represented as circles. The colour indicates whether it is incoming or outgoing. Black designates an outgoing connection and white indicates an incoming connection.

LogInterface represents an outgoing connection point and specifies two operations in the Log class that are crosscutting. This representation is similar to the representation of an interface. There are two significant differences between connection points and interfaces. The first is that connection points can be instantiated. The second is that connection points specify invocations to operations that they specified as corsscutting.

The relationhip between this specification and the Log class is represented through a <<binding>> relationship.

5.4.17.3.3 Integration specification

Behvaioral integration is specified in AspectJ style pointcuts. This apporach represents pointcuts in terms of connection points. The example presented in Figure 5-32 shows a pointcut representation. A pointcut is a classifier stereotyped as a <<pointcut>>. The classifier is represented as a compartmentalised structure. The top compartment holds the two types to indicate the two connection end points. The connection points are illustrated as connected white and black circles. The bottom compartment represents the actual integration specification. Pointcuts are represented as a signature like construct that is identical to the AspectJ pointcut language.

5.4.17.3.4 Specification of aspects

As illustrated in Figure 5-32, aspects are classifiers identified as an aspect through the <<aspect>> stereotype. The aspect representation includes connection points connected via dotted lines. This indicates that this is where two types are connecting and this is where advice is to be invoked. The aspect design element is compartmentalised. In the bottom compartment the advice and pointcut are associated. In the top compartments structurally crosscutting is represented.

5.4.17.3.5 Composition semantics

This representation is based on AspectJ. As such the composition semantics are similar to AspectJ.

5.4.18 Meta-models

5.4.18.1 Meta-models Method

Attempts have been made to extend the UML meta-model to support an AOD meta-model. We have categorised approaches based on the levels of abstraction that they support and the levels of concern separation that they facilitate. Here we discuss three approaches that alter the UML meta-model to support AOD.

5.4.18.2 Meta-models Artefacts

5.4.18.2.1 Meta-models AOD Language

Here we will describe each of the meta-models.

5.4.18.2.2 Specification of aspects

In [185, 186]a generic meta-model is described to support structural AO modelling. This meta-model is built around a set of core abstract elements and relationships. The core elements are crosscutting and base elements. The core relationship between these elements is a crosscutting relationship. The abstract crosscutting element describes the base semantics of the elements that are related to the asymmetric AOD model that this approach supports. Aspects are specified as specialized from crosscutting elements.

In the AspectJ meta-model [187] aspects are specified as an extension of a Java class. The meta-model is a precise model of the AspectJ language. Aspects model the same structure and behaviour as AspectJ aspects. The elements that compose aspects (pointcuts and advice) are represented as parts.

In the Hyper/J [188] aspects are specified as hyperslices. These hyperslices are represented as specialised java packages. Following the Hyper/J model, these packages contain a declaratively complete class structure.

5.4.18.2.3 Specification of crosscutting

In the generic meta-model a feature represents an operation or attribute. A crosscutting feature is a model element that describes a feature to be composed with one or more base elements. A crosscutting element specifies a design element in which crosscutting can be specified.

In the AspectJ and Hyper/J meta-models the specification of crosscutting reflects the AspectJ and HyperJ languages.

5.4.18.2.4 Integration specification

The generic meta-model specification of integration is modelled as a relationship that links crosscutting elements and base elements. This crosscutting or integration relationship is specified by the crosscutting interface, which models join points.

In the AspectJ meta-model integration, is specified through a pointcut model. The UML expression of a pointcut is similar to that of the pointcut construct in the AspectJ language.

In the Hyper/J meta-model, integration is specified in a hypermodule. A hypermodule is modelled as a specialisation of hyperslice that is related to a concern through an integration relationship.

5.4.18.2.5 Composition semantics

The composition semantics for a generic AOD meta-model are undefined. The composition semantics for the language specific AOD meta-models are similar to the languages that they model.

5.4.19 Aspect Oriented Software Development with Use Cases (AOSD\UC)

5.4.19.1 Aspect Oriented Software Development with Use Cases Method

Aspect Oriented Software Development with Use Cases (AOSDUC) is a use case driven software development method [38]. Use case diagrams represent concerns separately. This method provides a systematic process through which the concern separation inherent to use case diagrams is maintained through the software development lifecycle.

5.4.19.2 AOSD\UC Artefacts

5.4.19.2.1 AOSD\UC AOD Language

ASODUC represents a complete design method, that includes supports a transition from requirements to high level design to low level design and finally, implementation. To support all these phases AOSDUC provide a comprehensive design language.

5.4.19.2.2 Specification of aspects

At a high level of design, aspects are specified as use cases that extend other use cases. In more detailed design use cases are specified as packages. Within these packages use case design is represented.

Crosscutting use cases contain aspects. An aspect is a classifier that is identified through an <<aspect>> stereotype. An aspect classifier is graphically represented as a box that has two internal compartments. One of which contains pointcut declarations and the other contains class extensions.

Reusable aspects or utility use cases are specified as parameterised template packages. Parameters of the template are named and described within a box with a dotted edge on the top right hand side of a template package.

5.4.19.2.3 Specification of crosscutting

At a high level Use case specifications describe the extension flows that the flows specified for other use cases.

At a lower level of design crosscutting is specified in the class extension compartment of an aspect classifier at design.

Structural crosscutting is specified by declaring the class as in the class extensions compartment of an aspect classifier.

Behavioural crosscutting is specified by declaring an operation within a class, declared as a class extension. The signature of the operation is similar to that of an AspectJ advice signature. The operation names a pointcut that identifies where the behaviour crosscuts other use case slices. The type of advice is identified and the corresponding behaviour to be identified is also named.

5.4.19.2.4 Integration specification

Integration is specified at a high level in use case diagrams as an extends relationship.

Use case specification diagrams also describe the set of events at which integration of use cases can be specified and the flows that can be integrated at those points.

At lower level design, points of integration are specified through pointcuts. The pointcuts are similar to the pointcut construct in the AspectJ language both syntactically and semantically.

Integration is also specified in behavioural diagrams, including sequence diagrams, collaboration (or communication) diagrams and state chart diagrams. In sequence and collaboration diagrams the points at which the sequences should integrate is described. In state diagrams, state based integration is modelled.

5.4.19.2.5 Composition semantics

Composition semantics are based on AspectJ composition semantics.

5.4.19.3 AOSDUC Process

The AOSDUC process is a model driven iterative process.

During analysis, use case diagrams and use case specifications are defined and refined to provide an overview of the concerns in a system and the relationship between these concerns. The use case diagrams provide a high level view of the entire system and the specifications associated with each use case provide a detailed description of the behaviours that a use case represents, and where appropriate, the crosscutting nature of that behaviour.

The use case models contain actors, use cases and relationships between them. Actors represent a client of a system. Actors are associated with use cases. Use cases represent concerns. Include, generalisation and extend are the types of associations between use cases, described in use case diagrams. Include allows factoring out of common behaviours between use cases. Generalisation is an abstraction relationship between use cases similar to inheritance. The extension relationship indicates that the behaviour represented by an extending use case crosscuts a use case being extended. Use case diagrams are used to represent and characterise concerns that are found in requirements. In use case diagrams the use cases are modelled separately and the relationships between use cases are modelled.

The use case technique facilitates behavioural specification. The flow through a use case can be captured in a use case scenario. Use case scenarios are specified in a use case specification. A use case specification is a textual description of one or more event flows that occur when a use case is instantiated by an actor. These flows can be full or partial flows through the use case. Flows can be described as basic flows, alternate flows and sub-flows. Basic flows are the general case. Alternate flows are contingencies for when the general case does not apply. Sub-flows are event sequences that appear repetitively and are described in one place and referenced there after.

Use cases can be extended or can be used to extend other use cases. Extension points describe an event where use case behaviour is crosscut. Extension flows describe the flow that crosscut that event. Extension points and extension flows are specified in the use case specification.

Use case diagrams and use case behavioural specifications are defined separately. Use case diagrams are graphical representations that provide a visualisation of concerns and the inter-relationship between concerns. Use case scenarios are textual in nature and are used to describe the behaviour associated with use cases. To visualise the connection between the use case and the use case specification, use cases are depicted as a classifier in a box, with an ellipse in the top right hand corner of the box. Within this box the flows through the use case are named and the type for each flow is provided. Where the use cases are part of an extends relationship, the use case being extended can contain a description of the extension points at which it is extended. The extending use case can contain a description of the extension flows. Extension flows are named and described. The description of an extension flow contains the name of the extension point that it is to crosscut, how the extension flow is to affect the extension point and a description of the results that can occur after the execution of the extension flow.

During analysis a use case diagram is created to represent the entire system. From there, both the base and crosscutting behaviour is specified in use case specifications. Both the use case diagrams and associated specifications are defined and refined in an iterative process. Use cases represent slices of an application that can be developed separately and incrementally. In analysis, a set of domain concepts are identified for each use case. High level sequence and collaboration diagrams are used to model the sequence of interaction between domain concepts. These behavioural diagrams are based on use case specifications.

Once use cases, use case specifications, the domain concepts and high level sequence interactions are identified in analysis, design begins. Design is based on analysis. Use cases represent slices that can be designed separately. This approach supports a symmetric approach to AOD. For each use case or slice, a separate design model is constructed. The analysis phase in AOSDUC produces a high level design. In this approach once analysis is complete a platform specific design follows. Design as a process is specialised for the designing base use cases, crosscutting use cases and non-functional use cases.

To design base use cases, the designer begins by identifying components and component interfaces. Components correspond to domain concepts identified in analysis. In design, domain concepts may be designed as a class or design may require a number of platform specific classes to provide a design that can be implemented.

Each component exposes required and provided interfaces. Through creating a component diagram for a slice or use case, the high level design concepts can be expressed and the relationships between these design concepts can be captured as inter-component relationships. Once this high level view is provided, the use case can be depicted as packages that explicitly reference the use case being realised by name. Use case packages contain diagrams that provide detailed component specification. Class diagrams provide a structural design of the use case slice components. Each class represents some part of a component or a component itself. Sequence diagrams are used to model the interaction between the classes. Sequence and communication (formerly collaboration) diagrams are used to describe the flow of behaviour between classes and objects. The flow, in these diagrams, is based on the flows described in the use case specifications and high level interaction diagrams created during analysis.

Designing crosscutting use cases is similar to designing base use cases except that aspects also need to be modelled. Aspects are represented as classifiers, demarked with an <<aspect >> stereotype. Aspects are contained in the use case package. This classifier describes the crosscutting behaviour and structure associated with the use case. An aspect is a named classifier that contains pointcut declarations and class extensions. Pointcut declarations are based on the AspectJ pointcut model and language. Class extensions are depicted as class diagrams. The classes that are class extensions specify crosscutting behaviour and structure. Crosscutting behaviour or advice is depicted as an operation that is similar to the AspectJ advice model. To design an aspect the classes that specify crosscutting behaviour and structure are identified and crosscutting is be specified. Pointcuts that ensure the correct extension of related use cases are identified. Sequence diagrams that illustrate how behaviour is crosscut are created, to illustrate crosscutting behaviour.

Non-functional use cases are designed through the specialisation of utility use cases. A utility use case represents a non-functional concern that is crosscutting. To support reuse of non-functional crosscutting concerns, parameterised template packages can be used to represent utility use cases. Designing non-functional or utility use cases is similar to designing an aspect. To create a concrete aspect the template parameters are replaced arguments that are specific to the application. To reuse utility use cases, the designer must choose a utility use case that meets the non functional requirements and using arguments to create a concrete aspect.

Composition is modelled or specified at a high level in the use case diagrams, through include, generalise and extends relationships. Separate design models are specified during the design phase. The composition of design models is relative to those models being composed.

When design of a use case is complete, both the structural and behavioural diagrams created during design are used to implement the concern represented by the use case.

5.4.20 UMLAUT

5.4.20.1 UMLAUT Method

The UMLAUT framework [183] is a model transformation tool. From the point of view of Aspect Oriented Software Development, UMLAUT can be seen as a framework for building application specific weavers to weave multi-dimensional high level UML design models (functional, dynamic, deployment, and static aspects annotated with design pattern occurrences, stereotypes and tag values) into detailed design models suitable for either implementation, simulation or validation.

In addition to the manipulation of UML models, UMLAUT is able to manipulate any kind of models on any kind of repositories. A transformation can be run on any repository that has compatible meta models. The meta models are defined using the MOF (Meta Object Facility). UMLAUT is composed of a transformation language compiler and a framework of transformations written in this language. It allows complex model transformations far beyond MDA classical PIM to PSM mappings. A major idea that drove UMLAUT evolution is that a transformation is a kind of program so it must be possible to apply the MDA approach to itself. Experience shows that aspect oriented techniques are also useful to design transformations themselves, which are complex entities managing various concerns

Since 1998, UMLAUT has been used in the UML context to demonstrate several concepts, such as weaving design patterns, supporting the design by contract approach [185], weaving model aspects, generating code, generating test cases and interfacing with validation tools on the model.

UMLAUT transformation operations are written in the MTL transformation language.

5.4.20.2 UMLAUT Artefacts:

5.4.20.2.1 AOD language

UMLAUT supports aspect oriented design with libraries of reusable transformation operations. These operations include user-defined algorithms for identifying pointcuts in models and then applying modifications to the model. Specific AOD languages can thus be defined by building a specific framework, organized with design patterns (which are really design patterns at a meta model level).

UMLAUT includes some prototypes of AOD languages. A first prototype was designed within the QCCS European project (www.qccs.org). Pointcuts are defined using an UML extended template notation. The general weaving algorithm uses extended template parameter bindings to match model fragments. Specific state charts bound to model items such as operations provide advices for the weaving of the various behaviours gathered from the models fragments.

Because of its high degree of separation between model fragments and weaving advice, this AOD technique is symmetric in nature.

A graphically oriented technique for specifying pointcuts and advices was produced from QCCS' results [184]. UMLAUT supports another protocol oriented AOD technique. The associated AOD notation aims at formally defining the behaviour of a set of model fragments.

Each fragment's behaviour is defined by HMSC (hierarchical message sequence charts: automata of partially ordered event sets). HMSC makes it possible to specify model behaviours at any level of abstraction while retaining a clear semantics.

The weaver merges model fragments using a set of HMSC that build correspondences between events from the fragments' HMSC [186]. This technique provides compositions that have well-defined behaviours [187]. It is therefore suited to designs with fairly complex protocols between items, such as distributed architectures.

5.4.20.3 AOD process

Apart from the two AOD languages mentioned above, UMLAUT aims at supporting research and experiments on AOD languages and frameworks.

Through metamodel extensions and libraries of transformation operations, specialized designers can build domain specific aspect notations and weaving algorithms. Therefore the underlying AOD process identifies various designer roles and activities: users of aspects, designers of aspects, designers of weavers and aspect languages.

5.5 Comparison

5.5.1 Level of abstraction

Approach	Level of abstraction
Theme	Middle
AODM	Low-Middle
AAM	Middle-High
CoCompose	Middle-High
SUP	Middle
AML	Low-Middle
TranSAT	Low-High
AOCE	Low-High
UFA	Middle & High
ADT	Low-Middle
UXF	Middle
IDAM	Low
AVA	Middle-High
Component Views	Middle-High
Meta Models	Low
AOSDUC	Low-High
CAM/DAOP	Low-Middle
Activity	Middle
UMLAUT	Low-High

Table 5-7: level of abstraction

AODM identifies commonality between AOP approaches and use this as a basis for an AOD language design. Figure 5-2 taken from [153], illustrates the similarities between the AspectJ, Hyper/J and DemeterJ approaches to the separation of concerns. At Figure 5-2 A, we see the common specification of the crosscutting elements. Figure 5-2 C depicts the modularisation of the crosscutting elements. Figure 5-2 D depicts the common specification of integration. Figure 5-2 E shows that in all approaches the result is a composition. Finally, from Figure 5-2 F depicts the implementation of design. We consider AODM to be at the low-middle level of abstraction. This is because the AODM is abstracted over implementation but is not representative of architecture or requirements.

Theme is an analysis and design approach. The analysis part of Theme, called Theme/DOC, is based on requirements engineering and identifying requirements such that they can be designed using Theme/UML. Theme/UML is the design part of Theme. We have determined that Theme/UML sits at a middle level of abstraction. This is because Theme/UML focuses on the design view, allowing its mapping to Theme/Doc views to present the corresponding requirements view. Theme is also abstract enough to be independent of platform specific AOP. It has been shown that Theme/UML designs can be implemented on various AO platforms [160].

SUP supports a middle level of abstraction in that it is not based on any implementation platform. This approach suggests a set of stereotypes and does illustrate the use of state charts for aspect identification and aspect modelling. It does not provide a high level view of a system. It focuses on facilitating the separation of aspects concerns that crosscut one main base concern [165].

AAM supports high and middle levels of abstraction.

Firstly, aspects are described at a high level as "concern solutions". A concern solution is an abstract aspect that can be applied to a model to introduce behaviour and structure that realize or reuse the concern in a specific application or system. These high level aspects crosscut application or system boundaries because they are "context free". Context free refers to the fact that aspects, described at this level are not coupled with any one application. They can be applied to many applications. In other words, context free aspects don't crosscut an application, they crosscut an application domain.

Secondly, aspects are described at a middle design level. Middle level or "context sensitive" aspects are specializations of high level aspects. Context sensitive refers to the fact that the context of the system, to which the high level aspect is applied, is used to realize the high level aspect at a middle level design.

Concern modelling represents the highest level of abstraction that we have surveyed. Concerns in this approach are viewed as conceptual entities. No structural or behavioural specifications are offered in this approach to support AOD modelling. Concern modelling provides a concern taxonomy with which concerns can be described.

We classify **CoCompose** as a middle-to-high level design approach. CoCompose support the description of abstract aspects that can be applied to concrete designs.

UFA allows modelling of high level aspect that crosscut application boundaries. To reuse high level aspect models at middle level design the aspect is specialized to an application based on the application context. Our middle-to-high level classification of UFA is based on this.

We classify **UXF** as middle level design because it is independent of implementation platforms, but does not address high level design issues.

We characterise **AVA** as a middle-to-high level design approach. This characterisation is based on the fact that inter-aspect dependencies are considered a high level relationship. This approach supports modelling inter-aspect dependencies and also supports aspect design. The aspect design is platform independent and is supportive of middle level design.

As **AML** is coupled with AspectJ we consider the level of abstraction supported by AML to be low-middle.

ADT is a middle level design approach. This approach is independent of language but the approach allows the separation of only one type of concern.

AOCE is a low-to-high level AOD design approach. This is because the AOD supported in AOCE is not platform independent. AOCE also supports high level analysis.

TranSAT approach is a composition of individual approaches that are highly related, in that they have been developed by the same group, and follow from one another. At a high level there is the TranSAT (described in Sections 15.1 and 15.2) framework to support software architecture evolution. At a low level there is a UML notation to support design for the JAC AOP platform (described in Sections 15.1 & 15.3). As such we regard this work as a high and low level approach.

Component Views approach supports a middle-high level of abstraction. We consider this approach to be of a middle-high level of abstraction due to the focus on concern reuse across system boundaries.

Activity diagrams can be used at various levels of abstraction. Here they are being used to model crosscutting behaviour at a middle level of design.

The CAM/DAOP approaches use of MDA provides a potentially high level of abstraction. In [182] a methodology that stems form a middle level design to low level design is illustrated. MDA can be used to define concerns at high levels of abstraction.

Model Driven Architectures [189] have been proposed as a solution to this conflict between the need to provide high level and low level design [148]. The MDA framework has been used to separate AOD into a stepwise process in which design is expressed at different levels of abstraction in a hierarchy of languages that capture an abstract design and provide means for transformations to more concrete and detailed design [182]. Transformations between AOD languages have been investigated in [171] and in the MDA context in [190].

The level of separation supported by **IDAM** approaches is at the lower design levels. The approaches that we have surveyed here are both AspectJ centric.

Meta-model The first approach is a language independent meta-model which supports middle level design [185, 186]. The second meta-model supports, AspectJ centric, low level design [187]. The third, and final, meta-model supports low level, Hyper/J centric, design [188].

AOSDUC [38] recognises the problem of conflicting needs between high and low level of design, and provides a multi-level and model-driven view of design as a solution. Concerns and concern relations can be expressed at a high level of abstraction mitigating detail. Concerns and concern relations can also be expressed at a low level of abstraction with the high degree of detail required for design implementation. AOSD\UC primarily represents concerns and concern relationships in Use Case Diagrams. As such, we consider this approach to be a high level design method

UMLAUT [181] is based a framework for model-driven transformation, and therefore (like the CAM/DAOP approach) provides a potentially high and low level of abstraction.

5.5.2 Level of crosscutting

Approach	Level of concern separation
Theme	Symmetric
Aspect-Oriented Design Modelling (AODM)	Asymmetric
Aspect-Oriented Architecture Modelling (AAM)	Asymmetric
CoCompose	Symmetric
State charts and UML Profile (SUP)	Symmetric
Aspect Modelling Language (AML)	Asymmetric
TranSAT	Asymmetric
Aspect Oriented Component Engineering (AOCE)	Symmetric
UML for Aspects (UFA)	Symmetric
Aspects at Design Time (ADT)	Asymmetric
UML exchange Format (UXF)	Asymmetric
Implementation Driven Aspects (IDAM)	Asymmetric
Architectural Views of Aspects (AVA)	Symmetric
Component Views	Symmetric
Meta Models	Symmetric & Asymmetric
Aspect-Oriented Software Development with Use Cases (AOSDUC)	Symmetric
CAM/DAOP	Symmetric
Activity	Symmetric
UMLAUT	Symmetric

Table 5-8: level of concern separation

AODM supports an asymmetric AOD approach. AODM was originally AspectJ based, the AODM design model mimics the AspectJ programming language and composition model.

Theme constructs are more general than aspects, and more closely encompass concerns with relation to subject-oriented programming, a symmetric approach to separation at the code level [156]. Originally modelled on SOP, Theme also supports symmetric AOD. Crosscutting themes are characterized as “aspect themes”. Themes that are not crosscutting are termed “base themes”.

SUP supports the symmetric separation of concerns. Specifications for the separation and integration of base and aspect concerns are supported in the SUP approach.

AAM supports asymmetric separation of concerns. AAM models or concern solution models are described as orthogonal to a primary model. Using the concepts and mechanisms embodied in the AAM approach in a symmetric model is described in [191].

This view of concerns in **Concern modelling** is conceptual. We classify this approach to be symmetric as it is describes all concerns in as separate entities.

We characterise **CoCompose** as supporting a symmetric approach to AOD. This is due to the potential for the separation of both crosscutting and non-crosscutting concerns as features.

The ACM model is a symmetric model of AOD. **UFA, AOCE, Component Views and CAM/DAOP** are based on the ACM model facilitates symmetric modelling.

UXF: An aspect is considered to be a new classifier on the conceptual same level as a class. Crosscutting relationships between an aspect and class are also defined. This indicates that this approach is an asymmetric approach.

AVA Aspects can be crosscut by aspects on which they depend. Therefore, this approach supports symmetric AOD modelling.

The AML, IDAM and **ADT** approaches are based on the AspectJ language. As such, it supports an asymmetric model of AOD.

TrancSAT At a high level this approach supports a symmetric approach, while at the lower design level the UML notation supports an asymmetric model.

Aspects are represented as **activity diagrams** that can be used to model symmetric AOP.

Meta-models The language independent meta-model and AspectJ meta-models are both asymmetric models. In contrast, the Hyper/J meta-model is symmetric.

Although linked with an asymmetric implementation, the **AOSD\UC** approach supports symmetric modelling.

UMLAUT is a framework that enables design model transformation and composition. The AO transformations that the UMLAUT supports are based on UML extensions that describe the crosscutting nature of the design elements expressed in UML. Where UML extensions describe a symmetric model then symmetric composition is possible under UMLAUT.

5.5.3 Traceability

Some AOD processes that we have explored create and refine design models that represent requirements models and architecture models derived earlier in the software development lifecycle. Fully refined design models are those that include the detail required to directly implement designs on a particular platform.

Traceability is a property of a relationship between two models where one model is a refinement of another. Traceability is a measurement of the transparency or clarity of the refinement process.

The measures of traceability we can take when comparing AOD approaches are *external* and *internal*.

The external measure is one where we look at AOD models in relation to the full software development life cycle. External traceability is a measure of the transparency or clarity of refinement that an AOD approach provides between requirements or architecture models to design implementations.

Internal traceability is relevant where there are various phases within an AOD process. The design process consumes the models created during the requirements engineering and architecture development phases of the software development life cycle. These models are abstract and lack the detail required for the confident and unambiguous realisation of an implementation that meets requirements models and conforms to the architecture models. The internal phases of an AOD process systematically refine more abstract design models into design models that contain the detail required during implementation. Internal traceability is a measure of the transparency or clarity of refinement that an AOD approach provides between the phases of an AOD process.

Although we describe traceability as a measurable property it is difficult to measure as a statistical value due to the large disparity between the AOD approaches that are identified and described in this document. If all approaches were used in a common and unbiased design focused and detailed case study then a statistical comparison between approaches may be possible. However, in the absence of such a case study we will describe the potential for traceability facilitated in each approach.

Table 5-9 lists the approaches and briefly describes the external and internal traceability provided by each approach.

Approach	**Traceability**	
	External	**Internal**
Theme	Traceability from requirements to design and also design to implementation	Themes can be refined from abstract themes level to a class and method level design
AODM	Traceability from design to implementation	Traceability from system level design to specific constructs such as pointcuts
AAM	Traceability from architecture to design	N/A
CoCompose	Traceability from design to implementation	N/A
SUP	Traceability from requirements to design	N/A
AML	Traceability from design to implementation	N/A
TranSAT	Traceability from architecture to design and design to implementation	N/A
AOCE	Traceability from requirements to design and also design to implementation	Traceability from component level to class and method level
UFA	Traceability from architecture to design and design to implementation	Traceability from packages level to class and method level
ADT	Traceability scope limited to synchronisation	N/A
UXF/a	Traceability from design to implementation	N/A
IDAM	Traceability from design to implementation	

AVA	N/A	Traceability between sub-aspects and aspects
Component Views	Traceability from architecture to design	Traceability from component level to sub-component level
Meta Models	Traceability from design to implementation	N\A
AOSDUC	Traceability requirements to implementation	Traceability from use cases implementation focused design
CAM/DAOP	Traceability from design to implementation	Traceability derived through MDA transformations
Activity	N/A	Traceability of composition
UMLAUT	Traceability from design to implementation	Traceability derived through model transformations

Table 5-9 Traceability

Most of the AOD approaches that we have surveyed in this document are based on the UML. The UML provides a set of diagrams with well defined semantics. Each of these diagrams provides a view of a design model. There is a level of traceability between each of the system views presented in the UML diagrams. The approaches that are UML based all have this traceability inherently. It is where the UML is extended, either through the use of profiles or meta-model extensions, that the standard traceability facilitated by the UML is altered.

The **Theme** approach supports traceability throughout the software development lifecycle. Themes, identified during requirements engineering, can be traced across the software development life cycle to theme implementation.

Theme/Doc provides a process through which themes are identified in requirements. Theme/UML provides a means for separately refining theme requirements into UML diagrams that capture both the structure and behaviour of the themes. Theme/UML also provides semantics and mechanisms for describing how themes are to be composed.

Themes can be composed into composite themes. Composed themes represent a woven system. The composite can be traced back to the themes that have been used to create the composition. This is possible due to the well defined composition semantics defined in the Theme approach.

The Theme approach provides a series of guidelines for implementing themes on various AO implementation platforms. It is conceivable that these guidelines provide a certain degree of traceability from design to implementation and vice-versa.

In terms of internal traceability, the theme approach follows a process of continual refinement. Themes are initially represented as packages. Theme packages are decomposed into classes and that contain methods.

Themes are described in terms of their behaviour and objects in Theme/Doc. Theme designs are based refining these high-level requirements based descriptions. This process is a refinement process where by the high level descriptions are informally presented in loosely defined UML diagrams. These UML diagrams are them specialised, with additional detail until the themes come to the point that they can be implemented by the developer. The composition rules that designate how themes are to be composed are also refined in a similar way. As such the refinement process provides traceability from more abstract theme designs to more concrete and implementation focused design.

AODM supports external traceability from design to implementation. AODM was originally developed to support the representation of designs that targeted the AspectJ platform. The UML constructs provided by AODM are similar to AspectJ language constructs. As such, AODM supports traceability from design to an AspectJ implementation due to the direct association between the constructs available in both design and implementation languages.

Joinpoint designation diagrams are an extension of the original AODM approach. They provide a degree of independence from the AspectJ language constructs, in that they provide an AOP neutral way to represent pointcuts. These representations have been shown to be generic and provide a degree of traceability between the AODM approach and other AO implementation platforms beyond AspectJ.

Internal traceability is supported in AODM. AODM allows the designer to represent aspects, classes and the crosscutting association between the two. An aspect may have an affect on many heterogeneous classes at many different joinpoint shadows. To avoid clutter in the UML diagrams the associations between aspects and classes is expressed as a pointcut designator. The designer cannot easily determine from this representation, where exactly the joinpoints shadows exist, from inspecting the aspect or classes. AOSD provides a facility to model these joinpoint shadows. As such, there is traceability from declarations of potential crosscutting between aspects and classes.

AAM supports external traceability from architecture to design. In AAM aspects are defined at the architecture level. The behaviour, structure and crosscutting characteristics of the aspect is described free of application specific details. When designing an application the designer may decide that an aspect is required. The designer can then specialise the aspect into a design that can be applied to the specific application. There are a series of relationships that are used to specialise the architectural aspect. These relationships provide a basis on which the aspects usage can be traced from architecture to design.

CoCompose provides a design approach that automates the transformation from design to implementation. As such, this approach provides traceability between design and implementation. This is a non-UML based approach and does not support the internal traceability that the UML provides.

SUP provides guidelines for modelling crosscutting behaviour in state chart diagrams. SUP also proposes a UML profile that supports the use of specialised AO UML diagrams to model the crosscutting behaviour within aspects. The guidelines on how crosscutting behaviour should be modelled in design offers traceability between requirements and design.

The focus of **AML** is the automated generation of aspect source code based on UML diagrams. As such, AML supports traceability between design and implementation.

TranSAT provides traceability between architecture and design. Like the AAM approach, TranSAT supports the use of aspects expressed at the architecture level in design through specialising the aspect to the application context in which it is to apply.

AOCE supports both external and internal traceability. AOCE provides guidelines on how to take requirements and engineer the requirements for AOD. AOCE provides guidelines on how to realise these requirements in design also describes how the design is implemented. From these guidelines, a certain degree of traceability is possible from requirements to implementation.

AOCE is a component and aspect based approach. The design of components and the relationship between components are described. Components may be further decomposed into classes. This decomposition is based on satisfying the component interface and follows the standards associated with component-based design. There is potential for traceability in this decomposition.

UFA provides support for traceability from architecture level aspects being used in application specific design. Like TranSAT and AAM, UFA supports the expression of application independent aspects that can be specialised in design to specific applications. UFA targets LAC as an implementation platform and as such maintains a traceable relationship between UFA and LAC.

UFA, like the Theme approach, uses packages to encapsulate concern representations. These packages contain the diagrams that represent the structure and behaviour that describe aspects. Traceability exists from the higher level expressions of the aspect at the package level to the lower class level representations.

ADT is an early approach to representing aspects in design. The approach describes how synchronization as a crosscutting concern is separated at design in UML. Due to this constraint it is not possible to trace crosscutting concerns in the software development process outside of the synchronization concern.

UXF is another early approach to AOD. UXF prescribes a process of refinement for aspects where AO designs are implemented, executed and then tested. Based on the outcome the design is changed and the cycle begins again. Traceability is possible through the refinement process. The process is not well defined but does have potential for traceability.

The **IDAM** approaches facilitate traceability from AOD to AO source code. The IDAM approaches are AspectJ based and propose design approaches based on the AspectJ language. Because of the associations between the design language and the implementation language, potential for traceability between the two exists.

The **AVA** approach focuses on providing a means for describing dependencies between aspects during design. AVA provides a means to model the trace between dependant aspects.

Some AOD approaches are based on an implementation languages **Meta Model**. As such, the traceability that exists between design and implementation is increased. This is because there is a direct correspondence between the AOD and AOP languages.

Component Views support traceability from architecture to design. Like the AAM and TranSAT approaches, Component Views support the use (or reuse) of aspects expressed at the architecture level in design. This is achieved through specialising the aspect to the application context in which it is to apply. Aspects here represent reusable architecture. These aspects are re-used during application specific design. As such, the aspects can be traced from architecture to design.

The **AOSDUC** approach is a comprehensive approach to AOD. This approach promotes traceability in that the approach is use case driven. Use cases represent concerns that are designed and implemented separately. All artefacts that are related to use case can be traced back to a specific use case. Use cases are described by use case specifications derived from requirements. This gives traceability between design and requirements. The design can be refined from high-level component diagrams to package diagrams that express aspects as specialised classifiers. The approach provides guidelines for moving from the higher level component diagrams to the lower level package encapsulated diagrams. These guidelines provide a basis on which traceability, from higher level design representations to lower level design representations, can be derived. The design language that AOSDUC uses is AspectJ focused, sharing many constructs. Because of the symmetry between these AOD and AOP languages, traceability between the design representations and implementation is improved.

CAM\DAOP is an approach where traceability is well defined and well supported through the MDA transformations between models. The design models are refined from abstract models to implementation focused though a series of model transformations. A final transformation from design model to implementation is possible. As such, traceability from design to implementation is possible.

The **Activity** approach allows the modelling of crosscutting behaviour. Activity diagrams have changed in UML 2.0 to be more supportive of complex systems. Activity diagrams allow the modelling of complex concurrent behaviour. This approach provides a means for composing behaviours specified in activity diagrams. The composition of activity diagrams many be traced.

UMLAUT is based on model transformations, and provides traceability through transformations between models. Well defined transformation policies ensure that the output from a transformation can be traced back to the designs transformation input.

5.5.4 Composability

Composability is described in Section 2 as the ability to compose artefacts and consequently to view and understand the complete set of artefacts and their interrelationships, as well as to perceive the system as a whole from the associated artefacts.

The level of composability that an AOD approach supports has its basis on the level of concern separation supported by the approach. Section 5.5.2 describes characterises approaches as either supporting symmetric or asymmetric concern separation.

When concerns are represented in separated design modules, these designs may be composed into a unified design. Otherwise composition is deferred until implementation.

In most of the approaches we have looked at, UML extensions have been made to allow the designer to specify how models are to be composed. The extent to which models can be composed is based on the composition model supported by the AOD approach and composition semantics described for the approach.

Table 5-10 lists the approaches and briefly describes the composition specifications and the composition provided by each approach.

Approach	**Composability**	
	Composition Specification	**Composition**
Theme	Supports merge and override composition specification, together with binding of base elements to	Themes are composed into composite themes. A system is a composition of

	aspect templates. Theme has a well defined set of composition semantics that describe composition.	themes.
AODM	AODM is based on AspectJ and supports AspectJ specific composition specifications. Joinpoint designation diagrams extend AODM to support AOP neutral composition specifications. In both cases the composition semantics are influenced by the AspectJ composition semantics.	Composition is deferred until implementation. Join point indication diagrams and Join point composition diagrams for representing individual compositions.
AAM	Supports composition specifications that are similar to the Theme approach. Composition semantics are defined for the composition specification.	Composition of the aspect and core (or primary) model results in standard OO UML.
CoCompose	Composition specification and composition semantics are defined as implementation alternatives chosen through design algebra.	Composition is illustrated in a solution pattern but is deferred until implementation.
SUP	Composition is based on matching events that exist in independent state chart diagrams. Composition semantics are not well defined.	A composed system is not visualised
AML	Composition is specified in a connector model. The composition semantics are based on AspectJ.	Composition is deferred until implementation.
TranSAT	TranSAT is a framework in which composition specifications and composition semantics can be defined.	N/A
AOCE	Composition specification is done at the component level, composition semantics are not well defined.	Composition occurs at runtime and is visualised in collaboration diagrams.
UFA	Composition is specified in a connector model. The composition semantics are not well defined.	Composition results in standard OO UML.
ADT	Limited composition specification scope. Composition semantics are limited.	Composition in this approach is deferred until implementation.
UXF/a	Primitive composition specification. Composition semantics are not well defined.	Composition is not well defined.
IDAM	Composition specification and composition semantics are AspectJ based.	Composition in these approaches is deferred until implementation.
AVA	Allows composition dependency to be specified.	Composed AVA designs can be represented as standard OO UML diagrams.
Component Views	Provides a meta model which defines design elements to specify composition relationships.	Representation of composition is unclear.
Meta Models	The composition semantics of platform specific meta models are based on the platform on which they are based. Abstract meta-model does not provide strong composition specifications or	In the platform specific meta-models composition is deferred until implementation. In the more abstract meta model are not well defined.

	composition semantics.	
AOSDUC	Component based composition & AspectJ base composition specification and composition semantics.	Composition deferred until implementation.
CAM/DAOP	Multi design modal within an MDA framework. As design is specialised composition specifications become stronger, as do composition semantics.	Composition deferred until implementation.
Activity	Composition specified between behaviour expressed in activity diagrams.	Result is a new composite activity diagram.
UMLAUT	Multiple composition specifications can be used within the framework.	The result of a UMLAUT transformation is a woven set of designs.

Table 5-10: Composability

The **Theme** approach supports two forms of composition specifications, merge and override. Where an aspect theme is merged, base elements are bound to aspect template elements. The Theme approach provides a detailed set of composition semantics. The composition semantics describe how design artefacts are to be composed in accordance with the composition specifications. Override composition allows design elements in one theme to override (similar to the programmatic override) design elements in another theme. Merge composition supports the union of themes. Where conflicts exist in the composition of themes, resolutions of the conflicts can be specified. Themes can be composed into composite themes. Systems can be designed by designing themes that represent the systems concerns and composing the themes.

The composability provided by the **AODM** approach is based on AspectJ. As such the expression of composition specification design is based on the AspectJ language. The composition semantics associated with this model is also AspectJ based. AODM has been extended with joinpoint designation diagrams, which have increased the neutrality of the composition specification. The composition semantics for these diagrams are not provided in literature but it seems that the composition semantics are similar to those of AspectJ. Composition of the systems and the aspects that crosscut the system is not visualised in the AODM approach. AODM does however provide joinpoint indication diagrams. These diagrams allow the points designated by the composition specification to be represented diagrammatically. AODM also provides join point composition diagrams as a means of modelling composition of aspect advice and core design elements at joinpoint shadows.

The **AAM** model requires two levels of composability. The first is aimed at the contextualising aspects and the second is the actual composition specification. AAM supports the description of generic aspects in design. To apply the aspects to a specific design, the (abstract) aspect is made concrete by binding concrete application specific elements to the abstract aspect design elements. The concrete aspect or "context-specific aspect" can be composed with the primary or core model. The composition specifications are similar to those of Theme. Composition semantics are also defined for the composition specifications. In AAM, the result of composition is an OO UML diagram.

The **CoCompose** is a non UML approach, similar in concept to the AAM model. CoCompose supports the description of features. Features are similar to generic aspects that are firstly contextualised to a specific application design. The detail of how the concepts are to be composed is specified in implementation patterns. Implementation patterns specify how the features will be composed in an implementation. Implementation patterns are chosen through design algebra. The composition semantics are not well defined for CoCompose. Composition can be viewed in a solution pattern.

Composition specification in the **SUP** approach is based on matching event names in diagrams state chart diagrams. Composition is based on transmitting or broadcasting events between state chart diagrams. The composition semantics of this approach are not well defined.

The **AML** and **UFA** approaches support a connector model, where the composition specifications of designs are represented within a connector package. The composition semantics for AML are based on the AspectJ platform. The composition semantics for UFA are not well defined. Composition is deferred in AML until implementation (AML seeks to auto-generate AspectJ code from AML designs), while in UFA the composed model is standard OO UML.

The **TransSAT** approach is a framework to support composition. TranSAT does not provide composition specifications or composition semantics, allowing the designer to provide these when designing a system.

The **AOCE** approach is an aspect and component oriented approach where components specify the other components that they crosscut and how the component itself is to be crosscut. The composition semantics are not well defined and are dependant on the framework on which the components are implemented and deployed. Composition in component based systems is done at runtime. Composition is visualised in collaboration diagrams.

The **ADT** approach is limited to specifying the composition of synchronisation concerns. The composition semantics are also limited. Composition in this approach is deferred until implementation.

The **UXF** approach provides a primitive composition specification. Composition semantics are not well defined. Composition representation is not well defined.

The **IDAM** approach is based on AspectJ. The composition specification and composition semantics are based on the AspectJ platform. Composition is deferred until implementation.

The composition specification provided by the **AVA** approach is similar to that of the Theme approach. The AVA approach differs in that it provides a notation for specifying the order for the composition of designs representing crosscutting concerns. Basic composition semantics are provided by the AVA approach. Composed AVA designs may be represented as standard OO UML diagrams.

The **Component Views** approach provides a meta-model, the elements of which enable composition specification. The composition semantics are defined within this meta-model. It is unclear how composition is represented.

The **Meta Models** approach consists of three attempts to provide meta-models for aspect-oriented programming. Two of these meta models are platform specific - AspectJ and Hyper/J, the other is platform independent. For the platform specific models the composition specification and composition semantics are based on the underlying AOP platform. The platform independent meta-model provides an abstract "crosscutting element" within the meta model. Although the meta model provides a basis for describing the composition of design elements, composition specification is not well defined. The composition semantics for the generic model are not well defined.

The **AOSDUC** approach provides a means for specifying composition at a number of levels; the use case level, component level and sub-component level. The use case level is concerned with analysis. The component level design is based on the component composition specification and composition semantics of component based design. Sub-component level design is based on the AspectJ model. Due to the close tie between the design language and the AspectJ language it is our intuition that they share similar composition semantics. Composition in this approach is deferred until implementation.

The **CAM/DAOP** approach is an aspect and component oriented approach. This approach provides an MDA framework in which design models are specialised from abstract design representations. Each model provides a means for specifying component – aspect composition. On each model transformation a more specialised model is created. This model is more detailed and provides a greater level of detail for specifying composition. Abstract models contain weak composition semantics. As the model becomes more specialised and platform specific, the composition semantics become stronger. The models target a DAOP framework, on which designs are implemented. Composition is deferred until runtime within the DAOP framework.

The **Activity** approach is based on representing behaviour in design as Activity Diagrams. Activity diagrams are sequences of activities. Activity diagrams composition specification allows the addition or removal of activities from activity diagrams, as well as to join the activity sequences defined in separate activity diagrams. Composition can be represented in a new activity diagram that is based on the composed activity diagrams and the composition specification.

UMLAUT is a framework for model composition through model transformations. As a framework, UMLAUT allows for many different types of model transformations. As such, UMLAUT facilitates many different types of composition specifications to describe composition. The transformation of design models results in a woven design that is a composite of the designs that are integrated during transformation.

5.5.5 Evolvability

In Section 2, evolvability is described as the ease of changing artefacts in an existing design or the ease of addition or removal of artefacts.

The phrase "ease of changing artefacts in an existing design" covers a broad number of changes that can occur during design. Design is based on the outputs from the requirements and architecture phases of software development. Changes in these outputs may affect the design.

Design is also a specification for implementation. AOD approaches may be intended to be applicable to a number of implementation platforms or may target a specific platform. The degree to which an approach supports evolution may be significantly affected by the level of abstraction from the implementation platform that the approach supports. We have described the level of abstraction for each approach in Section 5.5.1. It is our intuition that AOD approaches that support higher levels of abstractions are more evolvable as they are less constrained by the restrictions that are enforced by AOD approaches that are platform specific.

The level of concern separation may also have a significant affect on the evolvability of an approach. AOD approaches that facilitate symmetric concern separation, encapsulate all concerns within separate design modules. The affects of change are localised to that particular module. In addition, the composition specification describing how concerns are to be integrated may be affected by a change to a concern. The composition specification is based on integrating concern designs and resolving any conflicts that exist between these designs. Where the design of one concern is altered, this may have an impact on the composition specifications that are dependant on the areas of the design that are altered. Where all concerns are separated, addition or removal of concerns is also relatively straightforward, requiring corresponding change to the composition specification.

Those AOD approaches that facilitate asymmetric concern separation, encapsulate crosscutting concerns within separate design modules, or aspects. Non-crosscutting concerns are not designed separately. Instead there is one core design and aspects are designed relative to the core. Within the core, non-crosscutting concerns are scattered and tangled, with corresponding evolution difficulties. Composition specifications describe how aspects are composed with the core. Significant changes to the core design may affect the aspects' composition specifications. Changes in aspect design are not as likely to require changes to the core. Addition or removal of aspects is straightforward. Addition or removal of core concerns requires invasive changes of the existing core.

Table 5-11 lists the approaches and briefly describes how each approach deals with changes in design and additions or removals from designs.

Approach	**Evolvability**	
	Change	**Addition - Removal**
Theme	Change to a theme may require composition rules and related themes to change.	Themes may be added or removed separately, requiring updates to the composition specification.
AODM	Changes in the core model may require changes in any aspect that are dependant on the points of change. Change is limited by the constraints of AspectJ.	Aspects can be added or removed without altering the core.
AAM	Changes in an application may require a change in the contextual relationship between context free aspects and applications.	Architectural aspects can be contextualised and added or aspects can be removed from the design.
CoCompose	Change in design can cause a need for a new implementation strategy.	Features can be added or removed.
SUP	Changes to state charts events may require changes to any behaviour that was triggered by those events.	Addition or removal of state charts can be done without affecting the other state charts that represent the remainder

		of the system.
AML	Changes in either the core or aspect are caught in a connector and limited by the constraints of AspectJ.	Addition or removal of aspects means that the relevant connector must also be added or removed.
TranSAT	Changes in an application may require a change in the contextual relationship between context free aspects and applications.	Architectural aspects can be contextualised and added or aspects can be removed from the design.
AOCE	Aspects are represented as components, change is localised within component.	Components are atomic modules that can be added or removed from the design.
UFA	Changes in either the core or aspect are caught in a connector.	Addition or removal of aspects means that the relevant connector must also be added or removed.
ADT	Change limited to synchronisation.	Addition and removal limited to synchronisation concerns.
UXF/a	Changes can be made to design in a distributed manner.	Additions to and removals from designs can be made in a distributed manner.
IDAM	Change is platform constrained.	Addition and removal are platform constrained.
AVA	Sub-aspect decomposition reduces evolution restrictions.	Sub-aspects that have no dependant aspects can be added or removed.
Component Views	Changes must be within the feasible View Components Meta-model.	Views can be added or removed.
Meta Models	Changes must be feasible within the (Open/AspectJ/HyperJ) Meta-models.	Addition or removal is based on the platform the meta model covers.
AOSDUC	Design is use case drive and change is isolated per use case.	Use case designs can be added ore removed from the design.
CAM/DAOP	Change is supported within the MDA models and transformations.	Additions can be made within the MDA models and transformations.
Activity	Changes in activity diagrams affect the activity diagrams that crosscut the activity diagram.	Activity Diagrams can be added and removed.
UMLAUT	UMLAUT allows automated design weaving. Changes in concern designs can be made and rewoven.	Concern designs can be added and removed. Designs can be rewoven to reflect the change.

Table 5-11: Evolvability

The **Theme** approach supports the symmetric and AOP platform independent design of concerns identified in requirements. A theme encapsulates the design related to a concern. Theme composition is described through composition relationships between themes. The affects of change to a concerns design are localised within a theme. This change may or may not affect the composition relationships between the theme and other themes with which it will be composed.

Themes are not dependant on one another. They are atomic representations of a concern in design. Because themes have no interdependence, themes can be added or removed without affecting other themes. Composition relationships are likely to change to incorporate the additions or removals.

The **AODM** was originally based on the AspectJ AOP language. Because of the close link between AODM and AspectJ, evolution of AODM designs may be restricted by the constraints imposed when mirroring the AspectJ language in design. Joinpoint designation diagrams have significantly weakened the association between the AODM and AspectJ, through providing a language independent way to model AspectJ pointcuts. This however does not represent a full abstraction and, as such, means that AODM as an approach is inherently evolutionary constrained because of its platform focus.

AODM supports an asymmetric model of design. Aspects can be changed, added or removed from the design with minimal affect on the core design. Aspects represent crosscutting concerns in design. The non-crosscutting concerns are represented in the core design. In the core concern design is scattered and tangled making change, addition and removal of concerns difficult.

The **AAM** asymmetric approach to AOD supports aspects designed independently of application context to be utilised in design. In this approach evolution is eased as pre-existing application-independent aspects can be contextualised for a specific application being designed. AAM does not focus on any particular platform and is not constrained to conform to any specific platform constraints.

CoCompose is a non-UML platform neutral, symmetric and high-level approach to AOD. CoCompose allows design to contain alternate implementation specifications at design. The design as such can evolve without platform constraints. Design elements that represent concerns can be added or removed with relative ease.

SUP is a symmetric, platform neutral, state chart driven approach to AOD. Separate state charts are created to model the state and behaviour of concerns. Events that trigger a state transition in one state chart are relevant to the state charts crosscut by that state chart. As concern behaviour is modularised within the state chart, change is localised within the state chart. Crosscutting events are matched by name. The changing of equivalent event names must be uniform across the state charts that describe a system to ensure consistency during evolution. State charts can be added or removed from the design without affecting the other state charts.

In **AML** and **UFA**, the core system design and aspect design are represented separately. The rules for the integration of the core and aspect are defined in a design module separate to that of the core and aspect design modules. As such both the core and aspects can be changed independently of one another. The change in either must be reflected in the connector, which acts as a level of indirection that shields both the core and aspects from changes in one another. The drawback is that any change that occurs in either the aspect or the core system must also be reflected in the connector. AML specialises the UFA approach toward the AspectJ platform, with corresponding constraints on evolution.

The **TranSAT** approach allows the addition and removal of crosscutting concerns in design. Like the AAM approach, application independent aspect designs can be integrated into core designs contextualising the concern. Aspects are removed by excluding an aspect design from the weaving instructions.

AOCE is a component-based approach. Components encapsulate concerns and may be crosscutting or non-crosscutting. Change is localised to a component. The addition and removal of components from design is facilitated.

The **ADT** approach is an early approach that is focused on representing synchronisation as an aspect in design. In this approach change, addition or removal of the design artefacts relating to the synchronisation concern are possible.

The **UXF** approach provides a model for AO UML transfer between repositories. This transfer facility provides a means through which designs can be changed, artefacts can be added to the design and artefacts can be removed from the design in a distributed manner.

IDAM approaches facilitate platform specific designs. These approaches are AspectJ centric and as such do not permit change outside of the AspectJ constraints.

The **AVA** approach is a symmetric, platform-neutral approach to AOD. This approach recognises that aspects are not completely independent of one another, providing a notation for expressing the dependencies between aspects. Evolution is supported, in that the designer can decompose aspects into sub-aspects. Sub-aspects can be composed into aspects that represent a crosscutting concern. Sub-aspects localise changes beyond the aspect making change in crosscutting concerns easier to handle. The AVA approach provides a notation to describe aspectual dependency. This notation allows the designer to inspect aspectual dependencies. From this, the designer can decide what course of action to take when removing an aspect from a design or when adding an aspect to the design.

The **Component Views** approach brings together components and viewpoints in a language neutral AOD approach. Like the AAM and TranSAT approaches the Component Views approach allows the reuse of application independent components in the design of an application. The approach facilitates application evolution through adding behaviour and properties to application components.

In **Meta models**, one meta model focuses on reflecting AspectJ and another focuses on reflecting Hyper/J . An abstract model provides a model for AOD that is not well defined and does not constrain evolution outside the scope of an implementation platform. In contrast, the AspectJ and Hyper/J approaches are constrained in the evolution that they can support as they are constrained by the AspectJ and Hyper/J platforms.

The **AOSDUC** approach is a use case driven approach in which each use case is designed and implemented as a separate slice of the system. Because of this use case driven separation, change within the specification of a use case is localised with the design associated with that use case. Use cases can be added or removed from the system design without affecting the system design.

The **CAM/DAOP** MDA driven approach provides a well-defined set of component based models organised in a specialisation hierarchy. There are well-defined rules for transforming the abstract models to concrete models. Evolutionary changes (including adding and removing design artefacts) can be made in abstract models and these models can be quickly and unambiguously transformed into more platform specific models.

The **Activity** approach allows the symmetric modelling of behaviour modelled in separate activity diagrams. In this model, changes to a concern's behaviour is localised within activity diagrams. New behaviour can be added to the design through the addition of a new activity diagram. Existing behaviours can be removed by excluding the corresponding activity diagram from an activity diagram composition.

UMLAUT supports design evolution, in that it provides a means to automate design composition or weaving. Concern designs can be altered, added or removed. The change in the design can be addressed by re-composing the concerns designs.

5.5.6 Scalability

Scalability is an important feature of any approach to design, indicating its ability to deal with designing large systems, as well as designing small systems.

UML is the standard language for object-oriented design. UML is also the basis for most of the AOD approaches discussed in this report. Most of the AOD approaches that are based on the UML describe extensions to the UML to handle aspect-oriented design. While the UML does provide useful separation capabilities (for example, the designer can separate various structural views from behaviour views), it was not designed to handle the kinds of concern separation that are the focus of AOSD. Standard UML designs therefore exhibit considerable crosscutting and tangling properties that AOSD is designed to avoid. As designs scale, crosscutting and tangling gets worse, negatively affecting evolution.

Component based design (CBD) helps reduce the size of a system's design. CBD supports the system design to be decomposed into components that isolate and encapsulate parts of a system. Components expose interfaces, through which the component can be used within the system. UML provides component diagrams to model components in design. Componentisation reduces the size of the core design by encapsulating the design within a component module usable within the core design.. Crosscutting concerns are not modularised with CBD.

In this report, we have characterised each AOD approach as being either symmetric or asymmetric. Asymmetric approaches facilitate separation of crosscutting concerns. Symmetric approaches facilitate separation of crosscutting concerns and non-crosscutting concerns. Separating concerns in design allows concerns to be modelled separately, with corresponding scalability benefits.

We have further characterised each approach in terms of the levels of abstraction the approach supports. Although designing concerns separately avoids the problems of scattering and tangling, it may be necessary to view the system as a whole, for verification purposes (for example). Some design approaches provide high-level views of the concerns separated in design and descriptions of how concerns will be composed. Low-level design is detailed and does not facilitate the designer achieving a holistic view of the system. Low-level design is necessary to provide a guide for developers during design implementation. Scaleable AOD approaches provide high-level views of system design as well as the low level views.

Table 5-12 lists the AOD approaches that we have investigated and briefly describes the scalability of the approach and also notes the proven usage of the approach in non-trivial cases.

Approach	**Scalability**	
	General	**Proven usage**
Theme	Theme allows a high level view of the system at a package level, themes are specialised separately.	Scalability illustrated in the design of a decentralised, mobile, multiplayer game.

AODM	Low-level AspectJ designs supported. Join point designation and indication diagrams provide higher level views. System level views are not provided.	Scalability unproven.
AAM	Provides a high level package view of the system.	Scalability unproven.
CoCompose	Allows solution patterns be applied to augment systems.	Scalability unproven.
SUP	Provides a means of representing aspects in relation to objects.	Bounded buffer example shows that scalability is limited.
AML	AML provides an AspectJ focused, connector model and allows integration of concerns to be specified within a connector.	Scalability unproven.
TranSAT	Provides a framework that promotes scalability.	N/A
AOCE	Component and aspect model.	Proven in the development of Serendipity-II, a large process management system.
UFA	Connector model that allows integration of concerns to be specified within a connector.	Scalability unproven.
ADT	Limited separation of synchronisation concern.	Not scaleable.
UXF/a	Focused on UML interchange.	Unknown.
IDAM	Provides low-level design and does not provide a high-level design perspective.	Scalability unproven.
AVA	Provides a means for modelling crosscutting concerns as multiple aspects in design.	Improves potential for scalability as aspects can be decomposed into sub-aspects.
Component Views	Component views provide high-level of the system at the package level.	Scalability unproven.
Meta Models	Language neutral and language specific meta models.	Scalability unproven.
AOSDUC	Provides a holistic view of the system with case modules. Use case modules contain lower level designs.	Scalability demonstrated in the design of a hotel management system.
CAM/DAOP	Scalability supported through a multi layer model of abstraction.	Scalability demonstrated in a the design of a virtual office application.
Activity	Limited to modelling behaviour.	Scalability unproven.
UMLAUT	Scalability supported through a multi-layer model of abstraction.	UMLAUT has been used since 1998 to perform a variety of composition styles.

Table 5-12: Level of Scalability

The **Theme** approach has illustrated its scalability in the design of a large-scale system. As described in [152], Theme was used to design a decentralised, mobile, multiplayer game, of large size. Each concern (non-crosscutting or crosscutting) was developed separately. Designs are composable into OO designs or can be implemented on AO platforms. Themes can be viewed at a high level as packages that are related through composition relationships. This view allows the designer to see how each theme contributes to the overall system.

The **AODM** approach is asymmetric and supports a lower level design based on the AspectJ platform. It also provides views with a higher-level perspective, in the form of joinpoint designation and joinpoint indication diagrams. AODM does not support high-level views that give a view of the overall system. AODM is illustrated through relatively trivial examples [145,147,148].

AAM, CoCompose and **Component Views** are design-focused approaches that allow application independent concerns to be expressed in design, and used in application specific scenarios. Although these approaches provide high-level views of how aspects are contextualised and integrated into design, they remain unproven in terms of being used in a large-scale project.

The **TranSAT** approaches a framework with a UML notation based on JAC**.** It promotes scalability by providing a means to represent and integrated concerns a high level.

SUP is a state chart driven approach to Aspect modelling in design, which is supported by a UML profile. This approach focuses on extracting aspects from core objects, illustrated with a bounded buffer example. SUP provides means to separate low-level behaviours. There is no higher-level system view provided. There is no evidence that this could be used in a large-scale project.

AML and **UFA** use a connector model where the integration between design models is specified in a connector. These approaches support a package level view where the concerns and their associated concerns are represented in packages. This provides a high-level view of the system, hiding the lower level details. UFA is a symmetric approach. The AML approach applies the UFA approach to the AspectJ platform. AML and UFA have not been used in a large-scale project.

The **AOCE** approach has illustrated its scalability in the design of a non-trivial process management tool, Serendipity-II [170]. The approach separates concerns into components and systemic components. These components and their interrelations are described at high component levels. Low-level component designs may be created separately.

The **ADT** approach is limited to separating synchronisation concerns during design. As such, the ADT approach is not scaleable.

The **UXF** approach is focused on the transfer of AO UML repositories between tools. This approach provides an AO meta-model. Non-trivial examples of the meta-model being used to create a design are not illustrated, and so Scalability cannot be determined.

IDAM and the AspectJ and Hyper/J **Meta-models** are low-level designs. These approaches do not provide high-level design models, and remain unproven in a large scale project.

The **AVA** approach provides a high level package based view of design, where the entire system can be viewed. This is similar to Theme but recognises that crosscutting concerns can be decomposed during design into sub-aspects. This approach provides a scaleable means for viewing crosscutting concerns in terms of the separate designs that represent the crosscutting concern and the composition dependencies that exist between these.

The **AOSDUC** approach has illustrated its scalability in the design of a non-trivial case study based on a hotel management system. This approach provides a means of presenting a holistic view of a system in use case modules diagrams. These use case module diagrams are specialised into separate designs that represent the structure and behaviour related to that use case.

The **CAM/DAOP** approach supports scalability by allowing the designer to specify design at various levels of abstraction. The CAM model is a high level view of the design that can be transformed into lower level designs that are closer to implementation. The approach has illustrated its scalability in the design of a large Virtual Office application.

The **Activity** approach is demonstrated through relatively trivial examples, one of which is an order processing system [143]. The activity approach supports behavioural models, but not structural modelling. As such, it does not allow the designer to view how changes in behaviour may alter system structure.

Like the CAM/DAOP approach, the UMLAUT approach focuses on model transformations as a means to compose concern designs. Through defining a layered model for transformations it is possible to support multiple levels of abstraction. Layering promotes scalability as high-level designs can represent a holistic view of the system to the designer hiding the more detailed designs. Where detail is required by the designer he or she can use one of the lower level design views. UMLAUT can potentially providing a multi-layered transformation framework for AOD.

UMLAUT has been used since 1998 and it has been proven through its usage in weaving design patterns, supporting the design by contract approach, weaving model aspects, generating code, generating test cases and interfacing with validation tools on the model.

6. Main Contributions of AO Analysis and Design

Having discussed and evaluated the contemporary non-AO and AO approaches to analysis and design, we have attempted to reveal their individual comparative strengths and weaknesses. Abstracting from the analysis of the individual approaches, we now attempt to provide a broader picture of the contributions that the aspect-oriented approaches provide for analysis and design.

6.1 Contributions of Aspect-Oriented Requirements Engineering

As it has been demonstrated in the discussion in section 3.3, the contemporary non-aspect-oriented requirements engineering approaches have been developed to primarily deal with one type of concerns. For instance, PREview (section 3.3.1.1) and NFRF (section 3.3.2.1) have underlined the importance of non-functional concerns and proposed means to ensure their fulfilment in a system. Problem Frames (section 3.3.3) and Use Cases (section 3.3.4), on the other hand, have focused on ensuring the required functionality of the system.

Aspect-Oriented approaches, such as Concern Modelling with Cosmos (section 3.4.4.1), and CORE (section 3.4.4.2), in contrast propagate the idea that all types of concerns are equally important and should be treated consistently, and non-discriminatively. *Thus, the first contribution of AORE is recognition of the need for equal treatment of functional and non-functional concerns.*

Further on, while some contemporary non-AO approaches (e.g. NFRF, PREview) have recognised that non-functional requirements are characterised by their broad influence on other requirements, they do not consider the similar broad influence of some functional requirements. Aspect-Oriented approaches (e.g., Theme/Doc, section 3.5.2.1, and CORE, section 3.4.4.2) have brought this fact to notice. *The second contribution of the AORE is the recognition that both functional and non-functional requirements can have a broad* ***crosscutting*** *influence on other requirements.*

Having acknowledged the importance of functional and non-functional crosscutting and non-crosscutting concerns, the AORE work has adopted the *early separation of concerns* principle: it should be possible to study each concern/requirement separately, on its own. In AORE the separation principle is complemented by the *composition* principle: it should be possible to compose each concern/requirement with the rest of the concerns/requirements of the system under construction to understand interactions and trade-offs among concerns.

Though most non-AO approaches have recognised that requirements have influence on each other, the issue of *Requirements-Level Composition* had not been investigated before AO. *Composability* – the support for combining individual requirements into coarser-grained requirements (as provided, for instance, by AORE with Arcade, section 3.4.1.1) – is the central notion of AORE. Using the AO terminology, this support should include a well defined *joinpoint model* and *composition semantics*. The *joinpoint model* exposes structured points through which requirements can be composed. The *composition semantics* provide systematic meaning to the composition.

Composability allows not only reviewing the requirements in their entirety, but also detection of potential conflicts very early on in order to either take corrective measures or appropriate decisions for the next development step. The composed requirements also become valuable sources of validation for the complete system [57]. *Thus, the* *third contribution of AORE is that of the notion and mechanism for requirement composability.*

A contribution related to the composition support is that of *trade-off resolution* in cases of conflicts and inconsistencies. This is not a new property in requirements engineering (for instance this is addressed as risk vs. cost analysis in the Viewpoints and Inconsistency Management approach, section 3.3.1.2). Nevertheless, *AORE has highlighted the possibility of trade-off identification though composition, the need for trade-off resolution and decision support, and underlined the importance of its systematic and traceable treatment; this is the fourth contribution of AORE.*

Another contribution of AORE is the provision of *mapping and influence detection support* of the requirement-level concerns to the concerns at later lifecycle stages. This again, is not a new concept in requirements engineering, but AO has revealed a new dimension here as well, by providing support for decisions as to if a requirement will map to another crosscutting artefact at the later stages, or will be absorbed by a decision, or turned into a local method/function, etc.

6.2 Contributions of Aspect-Oriented Architecture Design

Although the number of aspect-oriented architecture design approaches is limited, we can still infer several important contributions of aspect-oriented concepts at the architecture level. First of all, DAOP-ADL has shown the feasibility of using an aspect-oriented ADL. More importantly, the DAOP-ADL architecture descriptions can be reified for runtime manipulation hence supporting traceability of architectural choice to implementation aspects. The Perspectival concern space framework has demonstrated that multidimensional separation of concerns can provide additional support for evaluating software architectures. AOGA has shown how to identify aspects in the domain model and map these to architecture artefacts. In particular identifying aspects using feature diagrams might be of interest for designing aspect-oriented product line architectures. It should be noted that the software architecture design community has now an increasing consensus on the separation of the various architectural views. The represented aspect-oriented architecture design approaches could be used to enhance the different architectural design views with an aspectual view for representing the aspects.

There is still some work to do on the process level. The approaches that have been considered define a particular process but it is not clear yet how to generalise these altogether. However, the process provided in AOGA integrates generative and aspect-oriented approaches and could be of interest for product line engineering or even model-driven architecture design approaches.

There are no aspect-oriented architecture analysis approaches except ASAAM. ASAAM uses scenario-based approach to identify aspects during architecture design and could be adopted for an architecture redesign or refactoring process. ASAAM could also be applied to analyse the quality of an aspect-oriented architecture.

6.3 Contributions of Aspect-Oriented Design

As with systems in any programming paradigm, aspect-oriented systems need to be designed with good software engineering practices in mind. The design of a system is at least as important as the implementation itself, indeed, perhaps even more important.

As described previously in this document and elsewhere, systems that are *not* designed using aspect principles exhibit scattering and tangling properties that have considerable negative impact on good software engineering practices. These properties are manifest in many, if not all, stages of the development lifecycle. In particular for this section, significant benefits can be derived from applying aspect-oriented techniques to design artefacts.

In the infancy of aspect orientation, developers simply used object-oriented methods and languages (such as standard UML) for designing their aspects. This proved difficult, as standard UML was not designed to provide constructs to describe aspects: Trying to design aspects using object-oriented modelling techniques proved as problematic as trying to implement aspects using objects. Without the design constructs to separate crosscutting functionality, similar difficulties in modularizing the designs occur, with similar maintenance and evolution headaches. At a high level, the main contribution of aspect-oriented design has been *to provide designers with explicit means to model aspect-oriented systems*, deriving software engineering quality properties as a result.

In particular, this breaks down into a number of sub-contributions. Aspect-oriented design provides a means for the designer to reason about concerns (whether they are crosscutting or not) separately, and to *capture concern design specifications modularly.* In so doing, the system's design does not exhibit scattering and tangling properties that contradict software engineering quality principles. Where there is modularisation, there must also be a means to specify how those modules should be composed into the full system design. Aspect-oriented design provides a means to *specify how concern modules should be composed.* This includes both a means to specify how to compose concerns at a later stage of the development cycle, and also a means to compose concern design artefacts. In this manner, the designers can choose whether to move to an object-oriented or an aspect-oriented programming paradigm. When composing concern designs, or specifying how concerns should be composed at a later stage of the development lifecycle, it is likely that there are points of conflict or cooperation between some concerns to be composed. Aspect-oriented design provides a means to *specify how to resolve conflicts between concerns and to specify how concerns cooperate.* Such conflict or cooperation specifications will guide the composition process.

The design task of the development process supports and is supported by other tasks in the development process, such as requirements analysis, architecture design and implementation. As such, it is important that it is clear where design artefacts fit into this support structure. A significant contribution of aspect-oriented design is the extent to which there is *traceability of concerns to lifecycle stages both preceding design, and post design.* Such traceability increases the comprehensibility and maintainability of the system. In addition to traceability of concerns, aspect-oriented design *provides a mapping of the constructs used in design to those used by lifecycle stages both preceding design and post design,* further enhancing the traceability.

7. Emerging AO Analysis & Design Processes

The main purpose of this document is to carry out a survey of the current state of the art in the AO analysis and design (as presented in the earlier sections). However, this document is also intended as the basis for further work on development of an integrated AO analysis and design approach, synthesising the work of AOSD-Europe project partners. This section outlines the initial process models, for each stage of analysis and design, that have emerged from the discussion and comparisons earlier in this report.

7.1 Emerging Requirements Engineering Process:

As a necessary minimum, the future emergent requirements process must provide the most important features of Aspect-Oriented Requirements Engineering, as discussed in section 6.1, i.e.:

- Equal treatment of functional and non-functional requirements (or, more precisely all concerns);
- Identification and treatment of crosscutting requirements of both functional and non-functional type;
- Good support for requirement composition and trade-off resolution;
- Support for mapping and traceability to artefacts at later development stages.

An initial outline of such a process is presented in Figure 7-1.

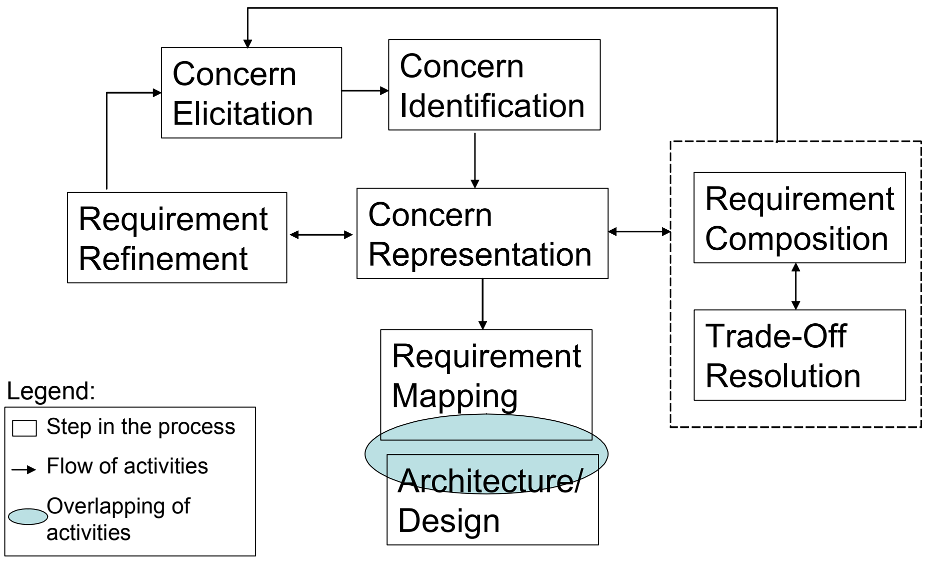


Figure 7-1 : The Emerging Aspect-Oriented Requirements Engineering Process.

The process commences with the *Concern Elicitation* step, where the requirements engineer establishes what are the issues of interest to the stakeholders, i.e. their *concerns* with respect to a given software system. This can be done through discussions with the stakeholders, interviews, ethnographic observation, etc. and complemented by guidelines such as the meta concern space described in section 3.4.4.2. The outcome of this stage is an initial list of broad stakeholder concerns (e.g., data retrieval, security, etc.).

This is followed by *Concern Identification* where the elicited concerns are elaborated (e.g., by using NFR catalogues we can define what does security concern imply, etc.). Tool support, e.g., the NLP-based concern identification tool or Theme/Doc (described in sections 3.4.1.1 and 3.5.2.1 respectively) can be very helpful in this regard. The identified concerns are represented at the *Concern Representation* stage (e.g., as goal and softgoal graphs, or viewpoints, themes, etc.).

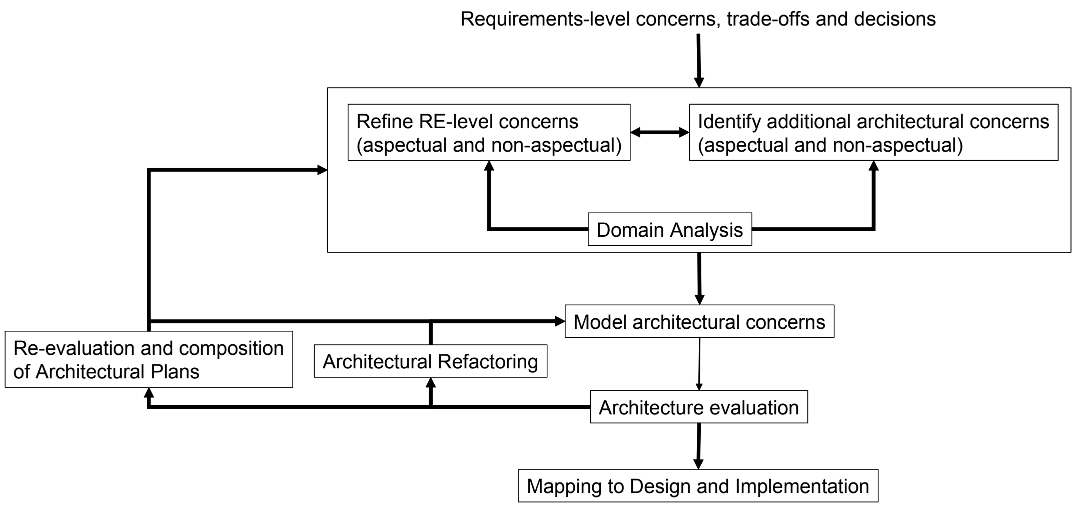
Simultaneously the concerns are *refined* and further elicited, identified and represented iteratively. When sufficient concerns are represented, the *Composition* stage gets initialised. At this stage concerns are composed and the outcome of the composition serves as the basis to identify conflicts between them. These conflicts are resolved through the *Trade-Off Resolution* stage, with the iteration cycle of refinement, elicitation, identification, and representation continuing all the while.

When a relatively stable set of requirement representations and compositions is achieved, the *Requirement Mapping* stage commences, where a set of guidelines is applied to help to turn requirements into architectural and design representations. The mapping onto architectural decisions should begin as requirements start to become clearer e.g., as in the TwinPeaks model [192] to ensure that the architectural choices reflect the stakeholders' concerns. This is essential because, as discussed in section 3.4.4.2, the concerns at the requirements level and their compositions drive the architectural choices pulling the architecture in various directions. It is essential that the final chosen architecture is at an optimal point with regards to the stakeholders' concerns.

7.2 Emerging Architecture Design Process

From the discussion of architecture approaches, we can observe an outline aspect-oriented architecture design process emerging. This process (shown in Figure 7-2) mainly draws upon AOGA, TranSAT and ASAAM. More specifically, the process:

- takes as input the requirements specification, requirements-level aspects and associated trade-offs as input;
- identifies additional architectural aspects (or refining existing requirements level aspects) via domain analysis;
- model the architecture, e.g., by using an approach such as DAOP-ADL, AOGA or TransSAT;
- utilises architecture evaluation, e.g., as in ASAAM, to identify additional aspects and undertake refactoring if needed;
- continuous re-evaluation and composition of new architecture plans based on previous architecture analysis and evaluation steps, e.g., as in TranSAT;
- mapping of the architectural decisions to design and implementation, e.g., as in methods such as AOGA and DAOP-ADL.



Requirements-level concerns, trade-offs and decisions
Refine RE-level concerns (aspectual and non-aspectual)
Identify additional architectural concerns (aspectual and non-aspectual)
Domain Analysis
Model architectural concerns
Re-evaluation and composition of Architectural Plans
Architectural Refactoring
Architecture evaluation
Mapping to Design and Implementation

Figure 7-2: The Emerging Aspect-oriented Architecture Process.

7.3 Emerging Design Process

As discussed in the *AO Approaches* section, some of the AOD design processes break design into separate tasks. For instance, the UFA and AML models separate design and specification, etc. Other approaches break design process into a number of phases. The best example of this is the CAM/DAOP approach. Here the high level design is elaborated into low level designs that can be implemented. Because with CAM/DAOP the design process is staged, there is an opportunity to specialise the design toward different implementation platforms.

Besides, we noted that some approaches separate design from composition (e.g., Theme/UML) while others do not support design model composition, and instead defer the composition until the implementation phase. We have also discussed that separating the design and composition into two tasks promotes reuse and provides the designer with a clearer view on the effects of composition.

With the knowledge of the above, we envisage an AOD process that separates the design process into tasks and phases, as well as allowing the designer to test their composition specifications through design-time composition. Our emerging design is described in Figure 7-3.

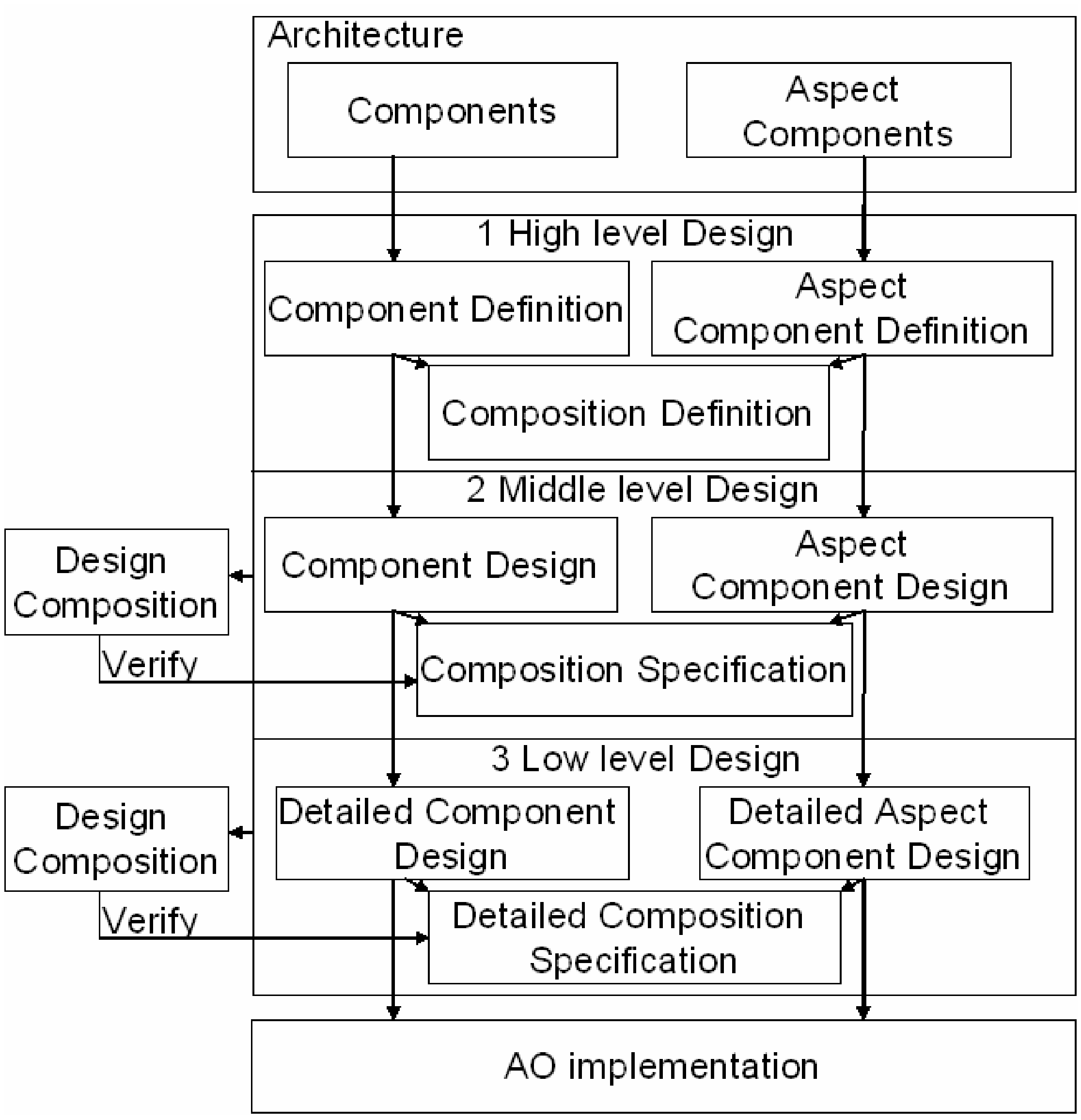


Figure 7-3: The Emerging Design Process.

Our process begins with refining the architecture into a high level design. There are three design phases - high, middle and low. High level designs are abstract and without detail, middle level designs are more detailed but platform independent, while low level designs are platform specific. During each design phase there are three things a designer needs to model – components, aspects and composition specifications. Components model concerns that are not crosscutting, and aspects model crosscutting concerns. Composition specifications describe how components and aspects are to be composed.

The design process is then a process of refinement between defined models of different levels. This process is intended as a "skeleton process" which can be specialised or extended to support different design models. Figure 7-4 illustrates how using the variations on the same process, different design models can be supported by our design process.

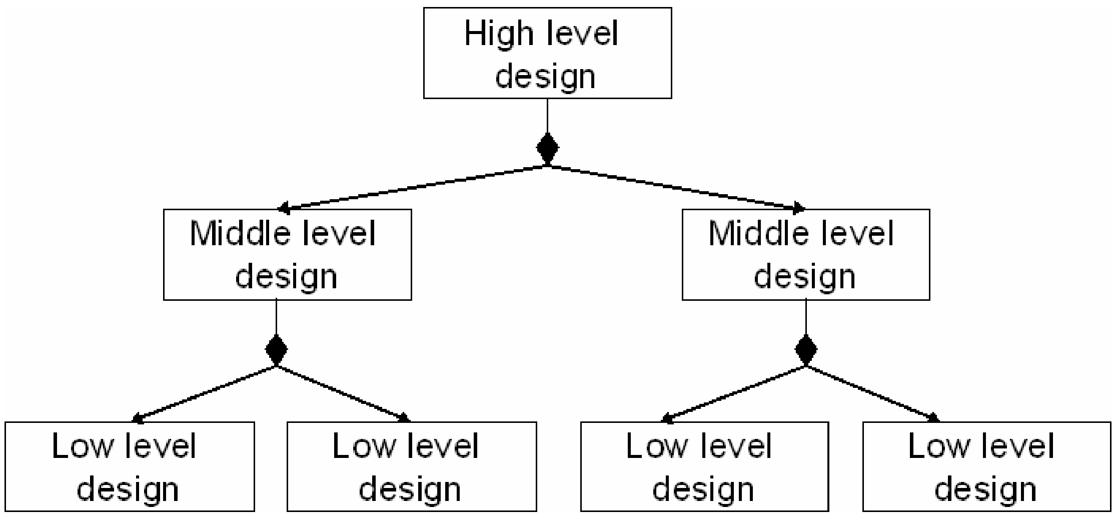


Figure 7-4: The Emerging Design Process.

8. Research Agenda to Be Addressed via Integration of Techniques

In the above section on *Emerging AO Analysis & Design Process* we have provided an outline of our initial integrated process. But what are the issues of immediate importance that we need to address in order to realise that process? It is this question that we address in the present section.

8.1 Outline of the Road to Integration – Requirements Engineering

It is fortunate that the main AO approaches to be integrated (namely Theme/Doc, AORE with Arcade, CORE) are complementary in a number of ways. While AORE with Arcade mainly focuses on broadly scoped (seemingly) non-functional concern treatment, Theme/Doc concentrates on behavioural (thus mainly functional) concern treatment. Besides, the latter approach applies after the requirements specification document is produced, while the former one can assist in its production. CORE, on the other hand, is focused on developing powerful composition and trade-off analysis mechanisms from a multi-dimensional perspective. The concern projection mechanisms in CORE as well as the guidelines on driving architectural choices from the projections and associated trade-off analysis can complement the treatment of non-functional and functional concerns in AORE with Arcade and Theme/Doc respectively.

An important question to be addressed by the integrated approach is: *can aspects help in requirement discovery from the very beginning of the RE process*? This question can be addressed through integration and use of the semantic analysis-based natural language processing work [59], which is based on AORE with Arcade, as well as the use of the meta concern space in CORE for eliciting concerns.

Following requirements discovery, the AORE with Arcade approach can be used to deal with requirements elaboration and non-functional crosscutting concern treatment (including composition and tradeoffs support). Possibilities of functional concern treatment with AORE with Arcade should also be investigated. Interesting insights can be provided here from the work on CORE; recall that CORE treats both functional and non-functional concerns in a uniform fashion and has powerful concern influence projection and trade-off analysis mechanisms. Following this, the first draft requirements specification document should emerge.

At this stage the Theme/Doc approach can be applied to identify any further crosscutting functional requirements and to improve the structure of the specification document. Theme/Doc also possesses visual modelling capabilities for requirements level aspects which can form a useful mechanism within an integrated AORE approach.

Once the requirements have been effectively represented, AORE with Arcade approach and CORE could be used again for making decisions about mapping crosscutting concerns to decisions, functions or aspects. Finally, the appropriate mapping to design can be done, producing either Theme-style, or other AO style designs.

We can observe from the above discussion that there are a range of complementary capabilities in the three main AORE approaches to be integrated. At the same time there are significant challenges in terms of the underlying models, e.g., multi-dimensional modelling in CORE compared to the two-dimensional base-aspect models in Theme/Doc and AORE with Arcade. Furthermore, though AORE with Arcade and CORE have similar composition mechanisms, the composition models differ greatly. The former is based on composing aspects with reference to a set of base viewpoints while the latter composes concerns in a multi-dimensional concern space. Theme/Doc, n the other hand, chooses to delay composition till the design stage. Traceability and mapping of the concerns within a requirements engineering process is also an interesting challenge due to the differing perspectives, foci and underlying representations used by the three approaches.

In summary, though we can observe some complementarities amongst the approaches to be integrated, there are also a range of interesting and challenging research issues to be addressed for such an integration to be effective. These research challenges will be one element of our future work (other elements being similar research challenges in integrating the AO architecture and design approaches) in the AOSD-Europe Analysis and Design lab.

8.2 Outline of the Road to Integration – Architecture Design

We can observe from the emerging aspect-oriented architecture design process in Section 7.2 that the various approaches from AOSD-Europe partners to be integrated, namely, DAOP-ADL, TranSAT, AOGA and ASAAM have a number of complementary features. For instance, AOGA and TranSAT can be employed for identifying architectural concerns and refining requirements-level concerns. DAOP-ADL, TranSAT and AOGA can all be used for architectural modelling. ASAAM and TranSAT can be used for architectural evaluation and informing refactoring decisions. Finally, AOGA and DAOP-ADL can be used to map architectures to detailed design and implementation and maintain traceability of architectural choices and decisions to the final system implementation. Similarly, some approaches, e.g., ASAAM and AOGA have Eclipse-based tool support.
At the same time, there are a number of interesting integration challenges. For instance, DAOP-ADL, TranSAT and AOGA each offer different architecture modelling mechanisms. Integration of these different modelling techniques is a significant challenge. It is helped by the fact that the modelling approaches are based on UML or its extensions. Nevertheless the different modelling abstractions employed pose significant integration bottlenecks to be addressed in the development of the integrated aspect-oriented architecture design approach in the AOSD-Europe Analysis and Design Lab. Similarly, ASAAM provides architecture evaluation capabilities but these are poorly related to the earlier architectural activities of concern identification and architecture modelling. The architecture composition mechanisms in the various approaches also differ considerably to pose interesting research issues during the development of the architecture composition mechanism for the integrated approach.

8.3 Outline of the Road to Integration – Design

As described in this document, there are many emerging approaches to aspect-oriented design, at varying stages of maturity. In general, they can be categorised as approaches that complement each other (i.e., provide for different requirements of aspect-oriented design), or approaches that overlap (i.e., provide for broadly similar requirements of aspect-oriented design in a different manner). In recognition of the emerging requirements for an overall aspect-oriented development methodology, integration of design methods must maintain a degree of flexibility as to the level of support for an aspect designer in the use of any available design method.

The road to integration should therefore include an investigation of the points of complement and overlap of each of the design approaches. Where there is complement, integration involves specification of heuristics and guidelines for combined usage to cover a maximum set of requirements for aspect-oriented design. Where there is overlap, integration involves specification of recommendations for further research into potential merging of approaches, together with guidelines for flexible usage of different approaches, as required by the aspect-oriented development methodology.

9. Conclusion

In this report we have surveyed a range of representative contemporary non-AO approaches to analysis and design and a comprehensive set of the significant AO approaches. We have discussed that, in many cases, the AO approaches have built on the strength of the non-AO approaches, but also aimed to address the previously overlooked issues of modularising crosscutting concerns.

We have discussed each presented approach based on a set of general criteria that reflects desirable properties of any software engineering approach, namely: traceability, composability, evolvability and scalability. From this we can conclude that generally *traceability* is improved due to modular representation of crosscutting concerns, and so are *evolvability* and *scalability*. The composability criterion, on the other hand, requires additional composition operators and procedures (e.g., design artefacts) when used with modularised crosscutting concerns, but also brings to light previously unexplored issues e.g., composition of requirements-level and architectural concerns and conflict detection through composition.

Further on, from the discussion of non-AO and AO approaches we have distilled the main contributions of the AO paradigm in the areas of requirements engineering, architecture design and detailed design.

For requirements engineering the main contributions are the equal treatment of functional and non-functional concerns, identification and treatment of crosscutting requirements of both functional and non-functional nature, provision of support for requirement composition and subsequent conflict detection and trade-off resolution, as well as support for mapping and traceability to artefacts at later development stages.

For architecture design the main contribution is the explicit treatment of crosscutting concerns during architecture modelling and evaluation, hence resulting in architectural choices that better reflect the stakeholders' decisions during requirements engineering and mapping and traceability of these choices to the detailed system design and implementation.

For detailed design these contributions mainly relate to support for modular representation of multiple types of concerns (e.g., non-functional, functional, crosscutting) and their composition, thus advancing such quality factors as traceability, understandability, maintainability, etc.

From this survey already some initial AO processes have emerged for requirements engineering, architecture design and detailed design stages of the software lifecycle. These are presented in section 7, yet it is clear that further research is needed into unification and integration of these processes into a complete AO analysis and design process. Such an integrated process will also form a valuable input into the integrated AOSD methodology to be developed as part of the *Atelier for AOSD* within AOSD-EUROPE.

We have also gained some initial insight into the challenges of integrating the work of our project partners. Some of such key challenges are reconciliation of differences between symmetric and asymmetric approaches, preservation of traceability within and across development stages and integration of multi-dimensional and two-dimensional approaches. These, along with other challenges, discussed in section 8 of this report will form the basis of our further work within the AOSD-Europe Analysis and Design Lab.

10. References

[1] A. Rashid, A. Moreira, and J. Araujo, "Modularisation and Composition of Aspectual Requirements," presented at 2nd International Conference on Aspect Oriented Software Development (AOSD), Boston, USA, 2003.

[2] Web Site: *Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, http://www.early-aspects.net/, URL: Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, maintained by A. Rashid, 2005.

[3] P. Sawyer, "Software Requirements," in *Software Engineering*, vol. 1, R. Thayer and M. Dorfman, Eds., 3 ed: IEEE Computer Society Press, to appear.

[4] I. Sommerville and P. Sawyer, "PREview Viewpoints for Process and Requirements Analysis," Lancaster University, Lancaster REAIMS/WP5.1/LU060, 29 May 1996.

[5] A. Rashid, "Website: Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design," URL: Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, 2005.

[6] A. Finkelstein and I. Sommerville, "The Viewpoints FAQ," *BCS/IEE Software Engineering Journal*, vol. 11, 1996.

[7] I. Sommerville and P. Sawyer, "Viewpoints: Principles, Problems and a Practical Approach to Requirements Engineering," *Annals of Software Engineering*, vol. 3, pp. 101-130, 1997.

[8] I. Jacobson, M. Chirsterson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, 4 ed: Addison-Wesley, 1992.

[9] A. Lamsweerde, "Goal-Oriented Requirements Engineering: A Guided Tour," presented at 5th IEEE International Symposium on Requirements Engineering, 2001.

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

[11] A. v. Lamsweerde, "Goal-Oriented Requirements Engineering: A Roundtrip from Research to Practice (Invited Keynote Paper)," presented at Requirements Engineering (RE 2004), Kyoto, Japan, 2004.

[12] I. Sommerville, *Software Engineering*, 7 ed: Addision-Wesley, 2004.

[13] A. Moreira, J. Araujo, and A. Rashid, "Multi-Dimensional Separation of Concerns in Requirements Engineering," presented at Requirements Engineering Conference (RE 05), Paris, France, 2005.

[14] A. Finkelstein and I. Sommerville, "The Viewpoints FAQ."

[15] P. Sawyer, I. Sommerville, and S. Viller, "PREview: tackling the real concerns of requirements engineering."

[16] B. Nuseibeh, J. Kramer, and A. Finkelstein, "A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification," *Transactions on Software Engineering, IEEE CS Press*, vol. 20, pp. 760-773, 1994.

[17] B. Nuseibeh, J. Kramer, and A. Finkelstein, "ViewPoints: Meaningful Relationships Are Difficult! Invited paper, Proceedings of International Conference on Software Engineering," presented at International Conference on Software Engineering (ICSE'03), Postland, Oregon, USA, 2003.

[18] E. Yu, "Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering," presented at Requirements Engineering, Washington D.C., USA., 1997.

[19] E. Yu, "Agent Orientation as a Modelling Paradigm," *Wirtschaftsinformatik*, vol. 43, pp. 123-132, 2001.

[20] E. Yu, "Modeling Strategic Relationships for process Reengineering." Toronto, Canada: University of Toronto, 1995.

[21] E. Yu, "Strategic Actor Modeling for Requirements Engineering," *Modelling Your System Goals - The I* Approach*. London, UK: British Computer Society -Requirements Engineering Special Interest Group, 2005.

[22] A. Dardenne, A. v. Lamsweerde, and S. Fickas, "Goal-Directed Requirements Acquisition.," *Science of Computer Programming*, vol. 20, pp. 3-50, 1993.

[23] M. Jackson, *Problem Frames: Analyzing and Structuring Software Development Problems*: Addison-Wesley, 2001.

[24] L. Barroca, J. Fiadeiro, M. Jackson, R. Laney, and B. Nuseibeh, "Problem Frames: a Case for Coordination," presented at International Conference on Coordination Models and Languages, Pisa, Italy, 2004.

[25] R. Laney, L. Barroca, M. Jackson, and B. Nuseibeh, "Composing Requirements Using Problem Frames," presented at Requirements Engineering Conference (RE 2004), Kyoto, Japan, 2004.

[26] M. Jackson, "A Discipline of Description," *Requirements Engineering*, vol. 3, pp. 73-78, 1998.

[27] I. Alexander and N. Maiden, *Scenarios, Stories, Use Cases Through the Systems Development Life-Cycle*: John Wiley & Sons, 2004.

[28] I. Alexander, "Misuse Cases: Use Cases with Hostile Intent," *IEEE Software*, vol. 20, pp. 58-66, 2003.

[29] I. Alexander, "Negative Scenarios and Misuse Cases," in *Scenarios, Stories, Use Cases Through the Systems Development Life-Cycle*, I. Alexander and N. Maiden, Eds.: John Wiley & Sons, 2004, pp. 119-139.

[30] I. Alexander and A. Farncombe, "Use and Misuse Cases in Railway Systems," in *Scenarios, Stories, Use Cases Through the Systems Development Life-Cycle*, I. Alexander and N. Maiden, Eds.: John Wiley & Sons, 2004, pp. 347-362.

[31] I. Sommerville, P. Sawyer, and S. Viller, "Viewpoints for requirements elicitation: a practical approach," presented at International Conference of Software Engineering (ICRE'98), Colorado Springs, Colorado, USA, 1998.

[32] S. Easterbrook and B. Nuseibeh, "Using ViewPoints for Inconsistency Management," *Software Engineering Journal, BCS/IEE Press*, vol. 11, pp. 31-43, 1996.

[33] B. Nuseibeh, S. Easterbrook, and A. Russo, "Making Inconsistency Respectable in Software Development," *Journal of Systems and Software*, vol. 58, pp. 171-180, 2001.

[34] L. Chung, "Dealing with Security Requirements During the Development of Information Systems," presented at Conference on Advanced Information Systems Engineering (CAiSE 93), Paris, France, 1993.

[35] A. v. Lamsweerde, A. Dardenne, B. Delcourt, and F. Dubisy, "The KAOS Project: Knowledge Acquisition in Automated Specification of Software," presented at American Association for Artificial Intelligence, Spring Symposium Series, Stanford University, 1991.

[36] E. Yu, J. Mylopoulos, and Y. Lesperance, "AI Models for Business Process Reengineering," *IEEE Expert: Intelligent Systems and Their Applications*, pp. 16-23, 1996.

[37] I. Jacobson, "Use Cases - Yesterday, Today, and Tomorrow," Rational Software, 2002.

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

[39] K. Allenby and T. Kelly, "Deriving Safety Requirements Using Scenarios," presented at 5th International Symposium on Requirements Engineering, Toronto, Canada, 2001.

[40] J. Araujo, A. Moreira, I. Brito, and A. Rashid, "Aspect-Oriented Requirements with UML," presented at Workshop on Aspect-Oriented Modelling with UML (held in conjunction with the International Conference on Unified Modelling Language UML 2002), 2002.

[41] Y. Yu, J. C. S. d. P. Leite, and J. Mylopoulos, "From Goals to Aspects: Discovering Aspects from Requirements Goal Models," presented at International Conference on Requirements Engineering, Kyoto, Japan, 2004.

[42] I. Jacobson, "Use Cases and Aspects—Working Seamlessly Together," *Journal of Object Technology*, vol. 2, pp. 7-28, 2003.

[43] J. Whittle, J. Araujo, and D.-K. Kim, "Modeling and Validating Interaction Aspects in UML," presented at AOSD Modeling With UML Workshop (located with UML 2003), San Francisco, USA, 2003.

[44] J. Whittle and J. Araujo, "Scenario Modeling with Aspects," *IEE Proceedings - Software*, vol. 151, pp. 157-172, 2004.

[45] J. Whittle and J. Araujo, "Scenario Modelling with Aspects," *IEE Proceedings - Software Special Issue*, vol. 151, pp. 157-172, 2004.

[46] A. Moreira, J. Araújo, and I. Brito, "Crosscutting Quality Attributes for Requirements Engineering," presented at Software Engineering and Knowledge Engineering Conference (SEKE), Ischia, Italy, 2002.

[47] J. Araujo and A. Moreira, "An Aspectual Use Case Driven Approach," presented at VIII Jornadas de Ingeniería de Software y Bases de Datos (JISBD), Alicante, Spain, 2003.

[48] S. M. Sutton, "Concerns in a Requirements Model - A Small Case Study," presented at Early Aspects 2003 Workshop: Aspect-Oriented Requirements Engineering and Architecture Design (held with AOSD 2003), Boston, USA, 2003.

[49] S. M. Sutton and I. Rouvellou, "Concern Modeling for Aspect-Oriented Software Development," in *Aspect-Oriented Software Development*, R. E. Filman, T. Elrad, S. Clarke, and M. Aksit, Eds.: Addison-Wesley, 2004, pp. 479-505.

[50] S. Sutton and I. Rouvellou, "Modeling of Software Concerns in Cosmos," in *Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002)*, G. Kiczales, Ed., 2002, pp. 127-133.

[51] A. Moreira, J. Araujo, and A. Rashid, "A Concern-Oriented Requirements Engineering Model," presented at Conference on Advanced Information Systems Engineering (CAiSE'05), Porto, Portugal, 2005.

[52] J. Grundy, "Aspect-Oriented Requirements Engineering for Component-based Software Systems," presented at 4th IEEE International Symposium on RE, 1999.

[53] J. Grundy, "Multi-perspective specification, design and implementation of software components using aspects," *International Journal of Software Engineering and Knowledge Engineering*, vol. 20, 2000.

[54] E. Baniassad and S. Clarke, "Finding Aspects in Requirements with Theme/Doc," presented at Workshop on Early Aspects (held with AOSD 2004), Lancaster, UK, 2004.

[55] E. Baniassad and S. Clarke, "Theme: An Approach for Aspect-Oriented Analysis and Design," presented at International Conference on Software Engineering, 2004.

[56] S. Clarke and E. Baniassad, "Theme: Aspect-Oriented Analysis and Design, URL: http://www.dsg.cs.tcd.ie/index.php?category_id=353," 2005.

[57] S. Katz and A. Rashid, "From Aspectual Requirements to Proof Obligations for Aspect-Oriented Systems," presented at International Conference on Requirements Engineering (RE), Kyoto, Japan, 2004.

[58] S. Katz and A. Rashid, "PROBE: From Requirements and Design to Proof Obligations for Aspect-Oriented Systems," Computing Department, Lancaster University, Lancaster COMP-002-2004, 2004.

[59] A. Sampaio, N. Loughran, A. Rashid, and P. Rayson, "Mining Aspects in Requirements," presented at Early Aspects 2005: Aspect-Oriented Requirements Engineering and Architecture Design Workshop (held with AOSD 2005), Chicago, Illinois, USA, 2005.

[60] P. Rayson, "WMATRIX," Paul Rayson, Lancaster University, URL: http://www.comp.lancs.ac.uk/ucrel/wmatrix/, 2005.

[61] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "Getting Started with AspectJ," *Comm. ACM*, vol. 44, pp. 59--65, 2001.

[62] P. L. Tarr and H. Ossher, *Hyper/J user and Installation Manual*: IBM Research, 2000.

[63] J. Whittle and J. Schumann, "Generating Statechart Designs from Scenarios," presented at International Conference on Software Engineering, Limerick, Ireland, 2000.

[64] R. France, D. Kim, S. Ghosh, and E. Song, "A UML-Based Pattern Specification Technique," *IEEE Transactions on Software Engineering*, vol. 30, pp. 193-2006, 2004.

[65] A. Moreira and J. Araújo, "Handling Unanticipated Requirements Change with Aspects," presented at Software Engineering and Knowledge Engineering Conference (SEKE'04), Banff, Canada, 2004.

[66] J. Mylopoulos, L. Chang, and B. Nixon, "Representing and Using Non-Functional Requirements: A Process-Oriented Approach," *IEEE Transactions on Software Engineering, Special Issue on Knowledge Representation in Software Development*, vol. 18, pp. 482-497, 1992.

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

[68] G. George, I. Ray, and R. France, "Using Aspects to Design a Secure System," presented at International Conference on Engineering of Complex Computer Systems, Maryland, USA, 2002.

[69] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. Sutton, "N Degrees of Separation: Multi-Dimensional Separation of Concerns," presented at Proc. 21st International Conference on Software Engineering (ICSE 1999), 1999.

[70] S. M. Sutton and I. Rouvellou, "Concerns in the Design of a Software Cache," presented at Workshop on Advanced Separation of Concerns (held with OOPSLA 2000), Minneapolis, USA, 2000.

[71] S. M. Sutton and I. Rouvellou, "Applicability of Categorization Theory to Multidimensional Separation of Concerns," presented at Workshop on Advanced Separation of Concerns in Object-Oriented Systems (held with OOPSLA 2001), Tampa, Florida, USA, 2001.

[72] W. Harrison and H. Ossher, "Subject-Oriented Programming - A Critique of Pure Objects," presented at Proc. 1993 Conf. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 93), 1993.

[73] E. Baniassad and S. Clarke, "Investigating the Use of Clues for Scaling Document-Level Concern Graphs," presented at Workshop on Early Aspects (held with ECOOP 2004), Vancouver, Canada, 2004.

[74] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke, "Viewpoints: A Framework for Multiple Perspectives in System Development," *International Journal of Software Engineering and Knowledge Engineering, Special issue on 'Trends and Future Research Directions in SEE', World Scientific Publishing Company Ltd*, vol. 2, pp. 31-57, 1992.

[75] J. Hall, M. Jackson, R. Laney, B. Nuseibeh, and L. Rapanotti, "Relating Software Requirements and Architectures using Problem Frames," presented at International Conference on Requirements Engineering (RE'02), Essen, Germany, 2002.

[76] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*: Addison-Wesley, 1998.

[77] M. Shaw and D. Garlan, *Software Architectures: Perspectives on an Emerging Discipline*. Englewood Cliffs, NJ: Prentice-Hall, 1996.

[78] P. C. Clements and L. M. Northrop, "Software Architecture: An Executive Overview," Carnegie Mellon University, Technical Report CMU/SEI-96-TR-003, 1996.

[79] M. Aksit, L. Bergmans, K. v. d. Berg, P. d. d. Broek, A. Rensink, A. Noutash, and B. Tekinerdogan, "Quality-Oriented Software Engineering," *Towards Software Architectures and Component Technology: The State of the Art in Research and Practice*, 2000.

[80] D. Garlan, R. T. Monroe, and D. Wile, "Acme: Architectural Description of Component-Based Systems," in *Foundations of Component-Based Systems*, M. S. Gary T. Leavens, Ed.: Cambridge University Press, 2000, pp. 47-68.

[81] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor, "Using Object-Oriented Typing to Support Architectural Design in the C2 Style," presented at Symposium on the Foundations of Software Engineering, San Francisco, CA, USA, 1996.

[82] N. Medvidovic, R. N. Taylor, and E. J. Whitehead, "Formal Modeling of Software Architectures at Multiple Levels of Abstraction.," presented at California Software Symposium, Los Angeles, CA, USA, 1996.

[83] A. Kompanek, "Proposed Aesop System Architecture," http://www-2.cs.cmu.edu/afs/cs/project/able/www/aesop/html/design_docs/aesop-new-design.ps, 1996.

[84] D. Garlan, R. Allen, and J. Ockerbloom, "Exploiting style in architectural design environments," presented at Symposium on Foundations of Software Engineering, New Orleans, Louisiana, United States, 1994.

[85] Web Site: *The Darwin Architecture Description Language*, http://www.doc.ic.ac.uk/~igeozg/Project/Darwin/, maintained by I. Georgiadis, visited May 2005.

[86] J. Magee, N. Dulay, and J. Kramer, "A constructive development environment for parallel and distributed programs," presented at Workshop on Configurable Distributed Systems, Pittsburgh, USA, 1994.

[87] Web Site: *Rapide™ Project*, http://pavg.stanford.edu/rapide/, Stanford University, May 2005.

[88] D. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*: Addison-Wesley, 2002.

[89] R. Allen and D. Garlan, "A formal basis for architectural connection," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. Volume 6, pp. 213 - 249, 1997.

[90] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J. E. Robbins, "Modeling software architectures in the Unified Modeling Language," *ACM Transactions on Software Engineering Methodologies*, vol. 11, pp. 2-57, 2002.

[91] L. Dobrica and E. Niemel, "A survey on software architecture analysis methods," *IEEE Trans. Softw. Eng*, vol. 28, pp. 638-653, 2002.

[92] R. Kazman, M. Klein, and P. Clements, "ATAM: Method Architecture Evaluation," Technical Report CMU/SEI-2000-TR-004 ESC-TR-2000-004, August 2000.

[93] N. Lassing, D. Rijsenbrij, and H. Vliet, "On software architecture analysis of flexibility, Complexity of changes: Size isn't everything," presented at Nordic Workshop Software Architecture (NOSA), 1999.

[94] G. Molter, "Integrating SAAM in Domain-Centric and Reuse-Based Development Processes," presented at Nordic Workshop Software Architecture (NOSA), 1999.

[95] C.-H. Lung, S. Bot, K. Kalaichelvan, and R. Kazman, "An approach to software architecture analysis for evolution and reusability," presented at Conference of the Centre for Advanced Studies on Collaborative Research, Toronto, Ontario, Canada, 1997.

[96] O. Bengtsson and J. Bosch, "Architecture Level Prediction of Software Maintenance," presented at Conference on Software Maintenance and Reengineering, Washington, DC, USA, 1999.

[97] P. Clements, "A Survey of Architectural Description Languages," presented at 8th International Workshop on Software Specification and Design, Paderborn, Germany, 1996.

[98] N. Medvidovic and R. N. Taylor, "A classification and Comparison Framework for Software Architecture Description Languages," *IEE Transactions on Software Engineering*, vol. 26, pp. 70-93, 2000.

[99] S. A. White, "Architectural Design Language Generation Project," presented at Workshop on Nasa Focus on Software Reuse, George Mason University, Fairfax, Virginia, USA, 1996.

[100] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*: Addison-Wesley, 1999.

[101] R. Prieto-Diaz and G. Arrango, *Domain Analysis and Software Systems Modeling*. Los Alamitos, California: IEEE Computer Society Press, 1991.

[102] H. Gomaa, "An Object-Oriented Domain Analysis and Modeling Method for Software Reuse," presented at Hawaii International Conference on System Sciences, Hawaii, USA, 1992.

[103] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, Technical Report CMU/SEI-90-TR-21, 1990.

[104] M. Simos, D. Creps, C. Klinger, L. Levine, and D. Allemang, "Organization Domain Modeling (ODM) Guidebook, Version 2.0," Informal Technical Report for STARS; http://www.synquiry.com STARS-VC-A025/001/00, June 14 1996.

[105] C. Czarnecki, "Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models," vol. PhD: Technical University of Ilmenau, 1999.

[106] G. Arrango, "Domain Analysis Methods," in *Software Reusability*, R. Schäfer, Prieto-Díaz, and M. Matsumoto, Eds.: Ellis Horwood, New York, New York, 1994, pp. 17-49.

[107] S. Wartik and R. Prieto-Díaz, "Criteria for Comparing Domain Analysis Approaches," *In International Journal of Software Engineering and Knowledge Engineering*, vol. 3, pp. 403-431, 1992.

[108] L. Bass, P. Clements, S. Cohen, L. Northrop, and J. Withey, "Workshop Report," presented at First Workshop on Product Line Practice, Pittsburgh, PA: Software Engineering Institute, USA, 1997.

[109] L. Bass, P. Clements, G. Chastek, S. Cohen, L. Northrop, and J. Withey, "2nd Product Line Practice Workshop Report," presented at Second Workshop on Product Line Practice, Pittsburgh, PA: Software Engineering Institute, USA, 1997.

[110] F. Hayes-Roth, *Architecture-Based Acquisition and Development of Software: Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program*, 1994.

[111] W. Tracz and L. Coglianese, "DSSA Engineering Process Guidelines," IBM Federal Systems Company, Technical Report ADAGE-IBM-9202, December 1992.

[112] C. Alexander, S. Ishikawa, and M. A. Silverstein, "Pattern Language," 1979.

[113] E. Gamma, R. Helms, R. Johnson, and J. Vlissdes, *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison-Wesley, 1995.

[114] J. O. Coplien, *Advanced C++ -Programming Styles and Idioms*. Reading, MA: Addison-Wesley, 1992.

[115] M. Fowler, *Analysis Patterns: Reusable Object Models*: Addison-Wesley, 1996.

[116] *Proceeding of the Pattern Languages of Programs Conference (PLOP 97)*. Monticello, Illinois, USA, 1997.

[117] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, "Pattern-Oriented Software Architecture: A System of Patterns," 1996.

[118] M. Shaw, "Making Choices: A Comparison of Styles for Software Architecture," vol. 12, pp. 27-41, 1995.

[119] G. Abowd, L. Bass, R. Kazman, and M. Webb, "SAAM: A Method for Analyzing the Properties of Software Architectures," presented at International Conference on Software Engineering, Sorrento, Italy, 1994.

[120] R. Kazman, G. Abowd, L. Bass, and P. Clements, "Scenario-Based Analysis of Software Architecture," *IEEE Software*, vol. 13, pp. 47-55, 1996.

[121] M. M. Kande, "A concern-oriented approach to software architecture," in *Computer Science*, vol. PhD. Lausanne,Switzerland: Swiss Federal Institute of Technology (EPFL), 2003.

[122] M. Pinto, L. Fuentes, and J. M. Troya, "DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development," presented at International Conference on GPCE, Erfurt, Germany, 2003.

[123] U. Kulesza, A. Garcia, and C. Lucena, "Generating aspect-oriented agent architectures," presented at Workshop on Early Aspects (held with AOSD 2004), 2004.

[124] U. Kulesza, A. Garcia, and C. Lucena, "Towards a method for the development of aspect-oriented generative approaches," 2004.

[125] O. Barais, E. Cariou, L. Duchien, N. Pessemier, and L. Seinturier, "TranSAT: A Framework for the Specifcation of Software Architecture Evolution," presented at Workshop on Coordination and Adaptation Techniques for Software Entities (held with ECOOP 2004), Oslo, Norway, 2004.

[126] B. Tekinerdogan, "ASAAM: Aspectual software architecture analysis method," presented at WICSA 4th Working IEEE/IFIP Conference on Software Architecture, 2004.

[127] A. Garcia, "From Objects to Agents: An Aspect-Oriented Approach," vol. PhD. Rio de Janeiro, Brazil: PUC-Rio, 2004.

[128] U. Kulesza, A. Garcia, C. Lucena, and A. v. Staa, "Integrating Generative and Aspect-Oriented Technologies," presented at 19th ACM SIGSoft Brazilian Symposium on Software Engineering, Brasília, Brazil, 2004.

[129] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools and Applications*: Addison-Wesley, 2000.

[130] OMG, "http://www.uml.org/," 2005.

[131] C. Chavez, A. Garcia, U. Kulesza, C. Sant'Anna, and C. Lucena, "Crosscutting Interfaces for Aspect-Oriented Modeling," PUC-Rio, Rio de Janeiro, Brazil, Technical Report TR-23-05, March 2005.

[132] O. Barais and L. Duchien, "SafArchie Studio: An ArgoUML extension to build Safe Architectures," presented at Workshop on Architecture Description Languages (WADL 2004), Toulouse, France, 2004.

[133] B. Tekinerdogan and F. Scholten, "ASAAM-T: A tool environment for identifying architectural aspects," Chicago 2005.

[134] L. Jingyue, S. H. Houmb, and A. A. Kvale, "A Process to Combine AOM and AOP: A Proposal Based on a Case Study," presented at Workshop on Aspect-Oriented Modeling (held with UML 2004), Lisbon, Portugal, 2004.

[135] P. Kroll, *The Rational Unified Process Made Easy, a practitioners guide to the RUP*: Addison Wesley, 2004.

[136] Nebulon, "http://www.featuredrivendevelopment.com/," 2005.

[137] S. Sendall and A. Strohmeier, "UML-based Fusion Analysis," presented at Unified Modeling Language: Beyond (UML) - the Standard, Fort Collins, CO, USA, 1999.

[138] D. D'Souza and A. C. Wills, *Objects, Components, and Frameworks with UML: The Catalysis(SM) Approach*: Addison-Wesley, 1998.

[139] F. Giunchiglia, J. Mylopoulos, and A. Perini, "The Tropos Software Development Methodology: Processes, Models and Diagrams," University of Trento, Technical Report DIT-02-008, 2001.

[140] C. Atkinson, J. Bayer, and D. Muthig, "Component-Based Product Line Development: The KobrA Approach," presented at Software Product Line Conference (SPLC), Pittsburgh, USA, 2000.

[141] K. Beck, *Extreme Programming Explained*: Addison Wesley, 1999.

[142] O. Aldawud, A. Bader, and T. Elrad, "UML Profile for Aspect Oriented Software Development," presented at Workshop on Aspect-Oriented Modelling with UML (held with AOSD 2003), Boston, Massachusetts, USA, 2003.

[143] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*: Addison Wesley Professional, 2000.

[144] G. Melby, "Using J2EE Technology for Implementation of ActorFrame Based UML2.0 Models," vol. Masters Thesis. Norway: Agder University, 2005.

[145] E. Barra, G. Genova, and J. Llorens, "An Approach to Aspect Modeling with UML 2.0," presented at Workshop on Aspect-oriented Modelling with UML (held with UML 2004), Lisbon, Portugal, 2004.

[146] J.-P. Barros and L. Gomes, "Activities as Behaviour Aspects," presented at Workshop on Aspect-oriented Modelling (held with UML 2002), Dresden, Germany, 2002.

[147] J.-P. Barros and L. Gomes, "Toward the Support for Crosscutting Concerns in Activity Diagrams: a Graphical Approach," presented at Workshop on Aspect-Oriented Modelling (held with UML 2003), San Francisco, California, USA, 2003.

[148] A. M. Reina, J. Torres, and M. Toro, "Towards developing generic solutions with aspects," presented at Workshop on Aspect-Oriented Modeling (held with UML 2004), Lisbon, Portugal, 2004.

[149] P. Desfray, "UML Profiles Versus Metamodeling Extensions: An Ongoing Debate," presented at Workshop on UML in the.COM Enterprise: Modeling Corba Components, XML/XMI and Metadata, Palm Springs, CA, USA, 2000.

[150] D. Stein, S. Hanenberg, and R. Unland, "Modeling Pointcuts," presented at Aspect-Oriented Requirements Engineering and Architecture Design workshop (held with AOSD 2004), Lancaster, UK, 2004.

[151] D. Stein, S. Hanenberg, and R. Unland, "Query Models," presented at Unified Modeling Language (UML) - Modeling Languages and Applications, Lisbon, Portugal, 2004.

[152] D. Stein, S. Hanenberg, and R. Unland, "A UML-based Aspect-Oriented Design Notation For AspectJ," presented at Aspect-Oriented Software Development (AOSD 2002), Enschede, The Netherlands, 2002.

[153] D. Stein, S. Hanenberg, and R. Unland, "Aspect-Oriented Modeling: Issues on Representing Crosscutting Features," presented at Workshop on Aspect-Oriented Modelling (held with AOSD 2003), Boston, Massachusetts, USA, 2003.

[154] D. Stein, S. Hanenberg, and R. Unland, "On Representing Join Points in the UML," presented at Workshop on Aspect-Oriented Modelling with UML (held with UML 2002), Dresden, Germany, 2002.

[155] D. Stein, S. Hanenberg, and R. Unland, "Designing Aspect-Oriented Crosscutting in UML," presented at Workshop on Aspect-Oriented Modeling with UML (held with AOSD-2002), Enschede, The Netherlands, 2002.

[156] S. Clarke, W. Harrison, H. Ossher, and P. Tarr, "Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code," presented at Proc. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 1999), Denver, Colorado, USA, 1999.

[157] R. E. Filman, T. Elrad, S. Clarke, and M. Aksit, *Aspect-Oriented Software Development*: Addison-Wesley, 2005.

[158] S. Clarke, "Composition of Object-Oriented Software Design Models," in *School of Computer Applications*, vol. Ph.D.: Dublin City University, 2001.

[159] S. Clarke, "Extending standard UML with model composition semantics," in *Science of Computer Programming*, vol. 44 Issue 1: Elsevier Science, 2002, pp. 71-100.

[160] S. Clarke and R. J. Walker, "Generic Aspect-Oriented Design with Theme/UML," in *Aspect-Oriented Software Development*: Addison-Wesley, 2005.

[161] S. Clarke and R. J. Walker, "Separating Crosscutting Concerns across the Lifecycle: From Composition Patterns to AspectJ and Hyper/J," Trinity College, Dublin, Technical Report TCD-CS-2001-15, May 2001.

[162] S. Clarke and R. J. Walker, "Towards a standard design language for AOSD," presented at 1st International Conference on Aspect-Oriented Software Development (AOSD 2002), 2002.

[163] O. Aldawud, T. Elrad, and A. Bader, "A UML Profile for Aspect Oriented Modeling," in *Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001)*. Tampa Bay, Florida, USA, 2001.

[164] O. Aldawud, A. Bader, and T. Elrad, "Weaving With Statecharts," in *Workshop on Aspect-Oriented Modeling with UML (held with AOSD-2002)*. Ensehede, The Netherlands, 2002.

[165] O. Aldawud, A. Bader, and T. Elrad, "Aspect-Oriented Modeling: Bridging the Gap between Implementation and Design," presented at Generative Programming and Component Engineering Conference (GPCE), Pittsburgh, PA, USA, 2002.

[166] C. Prehofer, "Graphical Composition of Components with Feature Interactions," in *Workshop on Aspect-Oriented Modeling with UML (held with AOSD-2002)*, 2002.

[167] R. France, I. Ray, G. Georg, and S. Ghosh, "Aspect-Oriented Approach to Early Design Modeling," *IEE Software*, vol. 151, pp. 173-186, 2004.

[168] G. Straw, G. Georg, E. Song, J. M. Bieman, S. Ghosh, and R. France, "Model Composition Directives," presented at Unified Modeling Language (UML) - Modeling Languages and Applications, Lisbon, Portugal, 2004.

[169] D. Wagelaar and L. Bergmans, "Using a concept-based approach to aspect oriented software design," presented at Aspect-Oriented Design workshop (held with AOSD 2002), Twente, Enschede, The Netherlands, 2002.

[170] S. Herrmann, "Composable Designs with UFA," presented at Workshop on Aspect-oriented Modelling (held with AOSD 2002), Enschede, The Netherlands, 2002.

[171] J. Suzuki and Y. Yamamoto, "Extending UML with Aspects: Aspect Support in the Design Phase," presented at Workshop on Aspect-Oriented Programming (held with ECOOP 1999), 1999.

[172] M. Katara, "Superposing UML Class Diagram," presented at Workshop on Aspect-Oriented Modeling with UML (held with AOSD-2002), 2002.

[173] M. Katara and S. Katz, "Architectural Views of Aspects," presented at Aspect-Oriented Software Development (AOSD 2003), Boston, Massachusetts, USA, 2003.

[174] I. Groher and T. Baumgarth, "Aspect-Orientation from Design to Code," presented at Workshop on Aspect-Oriented Requirements Engineering and Architecture Design (held with AOSD 2004), Lancaster, UK, 2004.

[175] I. Groher and S. Schulze, "Generating Aspect Code from UML Models," presented at Workshop on Aspect-Oriented Modelling with UML (held with AOSD 2003), Boston, Massachusetts, USA, 2003.

[176] I. Groher and S. Schulze, "Generating Aspect Code from UML Models.," presented at Workshop on Aspect-Oriented Modelling with UML (held with ULM 2003), San Francisco, California, USA, 2003.

[177] J. L. Herrero, F. Sanchez, F. Lucio, and M. Toro, "Introducing Separation of Aspects at Design Time," presented at Workshop on Aspects and Dimensions of Concerns (held with ECOOP 2000), 2000.

[178] R. Pawlak, L. Duchien, G. Florin, F. Legond-Aubry, L. Seinturier, and L. Martelli, "A UML Notation for Aspect-Oriented Software Design," presented at Workshop on Aspect-Oriented Modeling with UML (AOSD-2002), 2002.

[179] A. Muller, "Reusing Functional Aspects: From Composition to Parameterization," presented at Workshop on Aspect-Oriented Modeling with UML (held with UML 2004), Lisbon, Portugal, 2004.

[180] Z. Stojanovic and A. Dahanayak, "Components and Viewpoints as Integrated Separations of Concerns in System Designing," presented at Workshop on Identifying, Separating and Verifying Concerns in the Design (held with AOSD-2002), Enschede, The Neterlands, 2002.

[181] M. Pinto, L. Fuentes, and J. M. Troya, "A Dynamic Component and Aspect Oriented Platform," *The Computer Journal*, to appear.

[182] L. Fuentes, M. Pinto, and A. Vallecillo, "How MDA Can Help Designing Component- and Aspect-based Applications," presented at Enterprise Distributed Object Computing Conference (EDOC), Brisbane, Australia, 2003.

[183] W. Coelho and G. C. Murphy, "Modeling Aspects: An Implementation-Driven Approach.," presented at Workshop on Best Practices for Model-Driven Software Development (held with OOPSLA 2004), Vancouver, Canada, 2004.

[184] M. M. Kande, J. Kienzle, and A. Strohmeier, "From AOP to UML - A Bottom-Up Approach," presented at Workshop on Aspect-Oriented Modeling with UML (held with AOSD-2002), Enschede, The Netherlands, 2002.

[185] C. Chavez and C. Lucena, "A Metamodel for Aspect-Oriented Modeling," in *Workshop on Aspect-Oriented Modeling with UML (AOSD-2002)*. Enschede, The Netherlands, 2002.

[186] C. v. F. G. Chavez and C. J. P. d. Lucena, "Design-level Support for Aspect-Oriented Software Development," presented at Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001), Tampa Bay, Floriday, USA, 2001.

[187] Y. Han, G. Kniesel, and A. B. Cremers, "A Meta Model and Modeling Notation for AspectJ," presented at Workshop on Aspect-Oriented Modelling with UML (held with UML 2004), Lisbon, Portugal, 2004.

[188] I. Philippow, M. Riebisch, and K. Boellert, "The Hyper/UML Approach for Feature Based Software Design," presented at Workshop on Aspect-Oriented Modelling with UML (held with UML 2003), San Francisco, California, USA, 2003.

[189] OMG, "http://www.uml.org/mda/," 2005.

[190] S. J. Mellor, "A Framework for Aspect-Oriented Modeling," presented at Workshop on Aspect-Oriented Modelling with UML (held with UML 2003), San Francisco, California, USA, 2003.

[191] R. France, G. Georg, and I. Ray, "Supporting Multi-Dimensional Separation of Design Concerns," presented at Workshop on Aspect-oriented Modelling with UML (held with AOSD 2003), Boston, Massachusetts, USA, 2003.

[192] B. Nuseibeh, "Weaving Together Requirements and Architectures," *EEE Computer*, vol. 34, pp. 115-117, 2001.