

Semantics-based Composition for Aspect-Oriented Requirements Engineering

Ruzanna Chitchyan

Computing Department, Lancaster
University, Lancaster LA1 4WA, UK
+44-1524-510499

rouza@comp.lancs.ac.uk

Awais Rashid

Computing Department, Lancaster
University, Lancaster LA1 4WA, UK
+44-1524-510316

awais@comp.lancs.ac.uk

Paul Rayson, Robert Waters

Computing Department, Lancaster
University, Lancaster LA1 4WA, UK
+44 (0) 1524 510357

paul@comp.lancs.ac.uk

Abstract

In this paper, we discuss the limitations of the current syntactic composition mechanisms in aspect-oriented requirements engineering (AORE). We highlight that such composition mechanisms not only increase coupling between aspects and base concerns but are also insufficient to capture the intentionality of the aspect composition. Furthermore, they force the requirements engineer to reason about semantic influences and trade-offs among aspects from a syntactic perspective. We present a requirements description language (RDL) that enriches the existing natural language requirements specification with semantic information derived from the semantics of the natural language itself. Composition specifications are written based on these semantics rather than requirements syntax hence providing improved means for expressing the intentionality of the composition, in turn facilitating semantics-based reasoning about aspect influences and trade-offs. We also discuss the practicality of the use of this RDL by outlining the automation support for requirements annotation (realized as an extension of the Wmatrix natural language processing tool suite) to expose the semantics which are in turn utilized to facilitate composition and analysis (supported by the MRAT tool).

Categories and Subject Descriptors D.3.3 [Software Engineering]: Requirements/Specifications –languages, methodologies, tools

General Terms Design, Languages, Documentation

Keywords Aspect-oriented requirements engineering, requirements composition, expressive pointcuts, natural language processing

1. Introduction

Aspect-oriented requirements engineering (AORE) techniques aim to address the composability and subsequent analysis of crosscutting concerns during requirements engineering (RE). The goal often is to reveal aspect influences and mutual trade-offs among aspects before the architecture is derived. This offers early insights into architectural trade-offs providing opportunities for negotiation among the stakeholders before committing to specific architecture choices.

However, to date, aspect composition in AORE is based on syntactic references to requirements in the base, for instance,

direct references to requirement identifiers (e.g., as in Moreira et. al. [24] and Rashid et. al. [28]), use case steps and extension point labels (e.g., as in Jacobson and Ng [16]) or wild card matching based on such identifiers and labels [16, 24, 28]. This results in four problems:

1. Aspect compositions are specified on the basis of requirements structure rather than semantics of the requirements. As such the requirements engineer's (and the stakeholders') intentionality has to be mapped onto a syntax-governed structural model. Consequently, the intentional information is lost which has a detrimental effect on the subsequent analysis.
2. Use of syntactic composition mechanisms is cumbersome as they essentially require manual definition of each joinpoint individually before it can be used. One way to realize this is for the analyst to know ahead of requirements structuring and anticipate that a certain join point must be defined. This is done, for instance, in the use cases approach, where extension/inclusion use cases are often anticipated before they are defined (further discussed in section 2.1). Another approach for preparing joinpoints ahead is to provide a sub-requirement with a distinct id for each part/sentence of a requirement. This is the approach taken, for instance, in AORE with ARCADE (detailed in section 2.2). Alternatively, if the joinpoints are not prepared ahead, the analyst must re-check the set of requirements every time a new concern is composed, defining new joinpoints or separating new sub-requirements.
3. The trade-off analysis is conducted on the basis of these syntactic compositions from which semantic influences and interferences need to be inferred, requiring one to translate syntax-based identification of trade-off points to an analysis of semantic interactions among aspects.
4. Last, but not least, such syntactic composition models result in *fragile pointcuts* [17] where changes in the requirements syntax can invalidate the aspect compositions. This is analogous to the well-known problem of changes in classes affecting pointcut specifications or resulting in undesirable matching of joinpoints by pointcut expressions in languages such as AspectJ [1].

In this paper, we present a requirements description language (RDL) that addresses the above problems by enriching an existing natural language requirements specification with semantic information derived from the semantics of the natural language itself. Composition specifications are written based on these semantics rather than requirements syntax, hence providing improved means for expressing the intentionality of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AOSD 07, March 12-16, 2007, Vancouver Canada.

Copyright 2007 ACM 1-59593-615-7/07/03...\$5.00.

the composition, in turn facilitating semantics-based reasoning about aspect influences and trade-offs. Section 2 presents two motivating examples, one based on the AOSD with use cases approach by Jacobson and Ng [16], and the other based on the AORE with ARCADE approach by Rashid et. al. [28], demonstrating the limitations of current AORE composition models. Section 3 presents our RDL and the natural language semantics utilized by it as a basis of aspect composition. We also discuss how expressive pointcuts can be specified using the RDL's composition mechanism. Section 4 discusses the implementation of our semantics-based join point model and composition mechanism. We discuss extension of the Wmatrix natural language processing tool suite to expose the semantics. We also discuss the MRAT tool, which utilizes these semantics for composition and analysis. Section 5 revisits the four problems of syntactic composition models highlighted above and discusses how our RDL and its implementation address these. Section 6 provides an overview of the related work and Section 7 concludes the paper.

2. Motivating Example

In order to demonstrate the current practice of composition in AORE, we provide two motivating examples of aspect composition in two representative approaches from the state-of-the-art: Use Cases with Aspects [16] and AORE with Arcade [28]. We do not discuss examples based on Theme/Doc [5, 11] as it leaves composition to be addressed in design via Theme/UML [11] while our focus here is to demonstrate syntactic dependencies in requirements-level aspect composition mechanisms.

2.1 Use Cases with Aspects (UCA)

The Use Cases with Aspects (UCA) approach is illustrated in Fig. 1 using an extract of the hotel booking example discussed by Jacobson and Ng [16]. This approach modularizes the requirements into use cases, each of which describes an interaction between the actor and the software system for a certain service, as is done for the classic use cases approach [15]. Fig. 1(a) shows a part of a Reserve Room use case, whereby a user looks at the types and rates of the rooms available, and books a room. This use case describes an interaction for a specific service: booking the room. In this sub-section we will discuss two instances of advising this use case: one adding specific functionality to its given step(s) (sub-section 2.1.1) and the other adding a broadly scoped concern affecting this use case and other use cases (sub-section 2.1.2).

2.1.1 Advising a Single Use Case

Fig. 1(b) demonstrates an extension to the Reserve room use case: creation of a waiting list for a specific room type if there are no rooms available at the time of reservation. This is an *extension* use case, as the waiting list by itself does not provide any useful service to the actor (the user) but, as an add-on to the reservation service, it is valuable. As in the traditional use cases approach, the *extension* use case is essentially a part of the service of the extended use case, and is triggered from within a particular step of that extended use case. The link between these two use cases is established via definition of *Extension Points*. Such extension points are listed in a dedicated section of a use case with the same name, e.g., points E1 and E2 in Fig. 1(a), and refer to the numbered step of the use case at which the extension use case will be applied, as shown by the arrows marked as Links 1, 2.1 and 2.2 on Fig. 1(a).

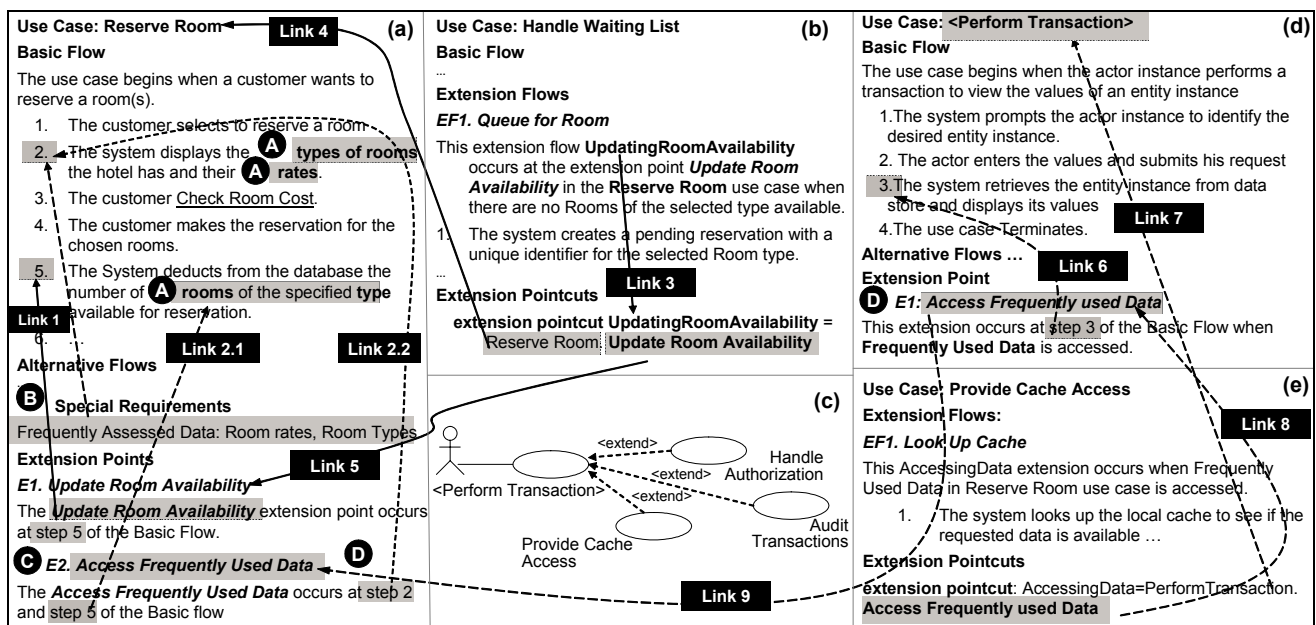


Figure 1: Pointcut definition in Aspect-Oriented Software Development with use cases

As shown in Fig. 1(a) the composition of the extension to the base flow is specified as “The *Access Frequently Used Data*

occurs at *step 2* and *step 5* of the Basic flow”. Already this small example demonstrates that the content of the use case has to be

structured into the corresponding steps and composition of the additional functionality is realized by referencing the ids of these steps (i.e., step 2, step 5), rather than their semantics, i.e. the content of the interaction taking place in a given step. Consequently, in order to understand what the above composition implies, one has to read both step 2, step 5 and maybe the *Access Frequently Used Data* extension flow.

To support composition of extension use cases, UCA introduces the notion of an *extension pointcut* defined within the extension use case (e.g., *UpdatingRoomAvailability* in Fig. 1(b)). The name of such an extension pointcut is referred to in the flow invoked from the correspondingly named *extension point* (e.g., *Update Room Availability*) in the base use case (e.g., Reserve Room in Fig. 1 (a)). Thus, the composition is realized by matching the name of the pointcut to that mentioned in the extension flow (e.g., Link 3 in Fig. 1(b)). The pointcut then refers to the base use case (Link 4 from Fig. 1(b) to 1(a)) and the extension point within it (Link 5 from Fig. 1(b) to 1(a)). The extension point, in turn, refers to the step number within the base use case (Link 1 in Fig. 1(a)).

The above referential complexity arises because a joinpoint (i.e. the named extension point) is created and referenced by name matching (via the extension pointcut). Though this preserves the so-called obliviousness¹ [13] for the base use case, it creates a strong syntactic dependency from the extension use case to the base use case. If the given extension use case were to be composed with more than one base use cases, the number of *prepared joinpoints* will correspondingly increase. The coupling between the extension and base use cases will also increase, as the exposed joinpoints and their use cases will be referenced one by one. Any changes to the ids of the use case steps or the labels used within the use cases will compromise the integrity of the composition and invalidate the pointcut specification. For instance, adding a new step after step 4 in Fig. 1(a) stating “The system checks submitted details for information omission” will invalidate the previous reference to step 5, which has now become step 6.

This fragility of the pointcut expressions is a direct consequence of syntax-based matching employed in the UCA approach. Semantics-based matching would have referred to the actual activity that needs to be advised, i.e. deducting the number of rooms from the database, rather than the unique id of the step at which it occurs. Such reference would have remained valid after the above “step addition” change.

2.1.2 Advising Multiple Use Cases

The joinpoint matching with extension pointcuts is even more complicated when composing non-functional properties. In order to deal with non-functional requirements (which cannot be represented using the traditional use cases approach), the UCA approach introduces the “Perform Transaction” template use case. This use case must reflect the functional effects of the special requirements which may potentially affect each step of every use case. For instance, in order to meet some response time requirements, the interaction time of each step for a use case must be controlled. As a result, it must bind to each

relevant step of the use case(s) of interest as well. A subset of such non-functional concerns that may extend the Perform Transaction use case is illustrated in Fig. 1(c). The main steps of the Perform Transaction use case itself are shown in Fig. 1(d) with its Provide Cache Access extension shown in Fig. 1(e).

Since Provide Cache Access is an extension to Perform Transaction, the joinpoint and pointcut correspondence is established in the same way as discussed above in Section 2.1.1:

- Joinpoint: by defining the *Access Frequently Used Data* extension point in the Perform Transaction use case which then refers to step 3 where the extension flow applies, as shown by Link 6 in Fig. 1 (d).
- Pointcut: by defining *AccessingData* extension pointcut in Provide Cache Access which refers to the base use case name (i.e., Perform Transaction) and the extension point (i.e., Access Frequently Used Data) in it, as shown by Links 7 and 8 respectively in Fig. 1(d).

However, since it is necessary to bind Perform Transaction itself to each relevant step of the affected “application” use cases, we also need to provide a way of identifying the affected use cases and their steps. In order to realize this, for each use case we have to:

1. check each step, to see if that step uses the data defined as “Frequently Accessed” (see points marked **(A)** in Fig. 1(a));
2. define a special requirement for the given use case, listing the types of data that are classified as “frequently accessed” (see Special Requirements Section marked **(B)** in Fig. 1(a));
3. define a named extension point which refers to each step in which the types listed in the Special Requirements section occur (e.g., Access Frequently Used Data extension point for steps 2 and 5 marked **(C)** in Fig. 1(a));
4. ensure that the named extension point from the previous step has the same name as the one used in the Perform Transaction Template (e.g., Link 9 from Fig. 1(a) to 1(d) marked **(D)** in the two figures). This ensures that when the Perform Transaction template is bound to a given use case the correct extension point is matched by the given extension pointcut.

Thus, the above complicated procedure is aimed at capturing a composition which with use of semantics can be reduced to “Use the cache whenever ‘frequently accessed data’ is accessed”. Yet, in UCA, the pointcut specification for this composition with a list of all affected use cases and extension points is so cumbersome that even the authors of the UCA approach avoid defining it. Appreciating the complexity of such composition, the authors suggest that “...you need not always define extension points during use case modeling... often it is more fruitful to define them during analysis and design...” [16, pp. 98-99]. However, deferring such extension point definitions to later stages results in poor appreciation of the affect of non-functional concerns that can in turn lead to poorly motivated architectural choices [23, 24, 26].

2.2 AORE with ARCADE

In AORE with ARCADE the main requirement decomposition units are viewpoints and aspects, where aspects encapsulate requirements that crosscut the viewpoint decomposition. An example of an ARCADE viewpoint and aspect is shown in Fig. 2 (a) and (b) respectively. Each of the viewpoints and aspects

¹ Note that we consider obliviousness to be a generally desirable though not fundamental property as discussed by Rashid and Moreira [27]. Obliviousness in this case contributes to the fragility of the pointcut expressions.

has a unique name and encompasses a set of requirements and sub-requirements. Each requirement has a unique identification number within the scope of its enclosing viewpoint or aspect (Fig. 2 (a) and (b)).

The corresponding ARCADE composition is shown in Fig. 2(c). It states that cache access functionality specified in the CacheAccess aspect (i.e. all the cache access requirements) should be provided for the room reservation and rate viewing requirement (id=1) of the Customer viewpoint. Fig. 2(c) demonstrates that, in an ARCADE composition, requirements and aspects are referenced via their unique names and ids within their defining scopes. Thus, here, as in the UCA approach, the requirements are prepared for composition by initially structuring them in such a way that any given requirements statement of interest for composition is given a separate id. The pointcuts are entirely syntactically defined by enumerating the unique “name.requirement_id” pairs (though some simple quantifiers, such as “all” can also be used, e.g., Fig. 2(c)). This leads to composition fragility as addition or removal of requirements may alter requirements identifiers resulting in undesirable matching of requirements based on the composition specification. Even if we assume that requirements identifiers are immutable and non-reusable (a hard constraint), the use of quantifiers such as “all” can still match newly added requirements or continue to attempt to match removed requirements hence compromising the integrity of the requirements analysis.

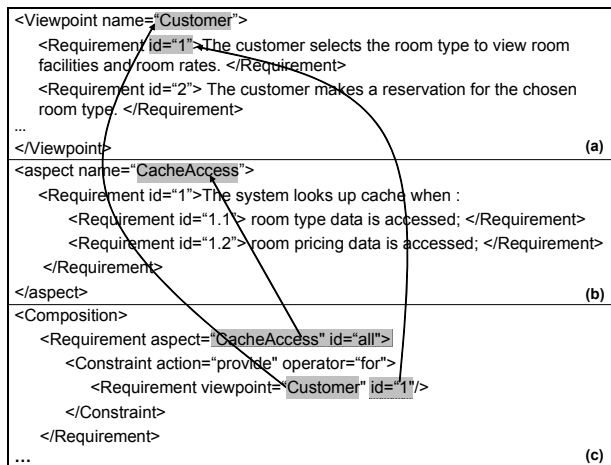


Figure 2: Composition with ARCADE

Furthermore, the semantics of the composition have to be mapped onto a set of name.id pairs for the relevant requirements. Reading and understanding such a composition specification is not an easy task, as one needs to constantly refer back to the corresponding viewpoint/aspect to see what a requirement with a particular id meant. The goal of the composition in this case is to reveal potential interactions among requirements-level aspects, but the composition mechanism forces one to reason syntactically about such semantic dependencies. Consequently, the requirements engineer has to elicit the semantic dependencies from the syntactic composition and any tool support is limited to inferring syntactic interactions among aspects.

Here too, as for UCA, it would be much more intentional to express the composition based on its semantics, saying “Use the cache whenever ‘frequently accessed data’ is accessed”. Instead, as described earlier, this composition is stated in terms of “all requirements” of CacheAccess and “requirement 1” of Customer, etc.

2.3 Limitations of Current AORE Composition Mechanisms

The discussion in sections 2.1 and 2.2 reveals that current AORE approaches:

- syntactically define each joinpoint by providing a unique name/id for it;
- explicitly identify and structure the requirements that will be affected by aspects so that they can be pinpointed and modified by referencing their ids. This is necessary as otherwise there is no way of separating the part of the requirement with a given id affected by an aspect from the non-affected part. This implies that either the requirements must be uniquely identified sentence by sentence (similar to ARCADE approach), or any new aspect/composition may force requirements re-structuring (similar to the UCA approach).
- specify pointcuts by explicitly listing the names/ids of the joinpoints they need to match or using quantification mechanisms (e.g., “all”, “include sub-requirements” in ARCADE, or listing several use case step numbers in one extension point as in UCA), which leads to pointcut fragility;
- require modular and compositional reasoning (i.e. reasoning about the global properties of the system [27]) about aspects and their mutual dependencies (as well as their dependencies on the base) based on syntactic matching of joinpoints in pointcut expressions.

In the following sections, we discuss how such composition fragility can be overcome by making the natural language semantics of requirements more explicit and basing the compositions on these semantics. We also discuss how this facilitates reasoning about semantic dependencies and interactions amongst aspects for the requirements engineer as well as for dependency-inference tools.

3. RDL

In our work we use the richness of the natural language to support expressive semantics-based pointcuts within a requirements description language (RDL). We utilize the fact that the natural language itself has a clearly defined set of syntactic rules and semantic elements, precise enough to support definition of a flexible composition mechanism for requirements analysis. Though a more formal semantics would undoubtedly be more precise, the natural language semantics capture the stakeholders’ needs as expressed in the elicited requirements and hence are more suited to aspect-oriented requirements analysis. Our RDL annotates the syntactic elements of the natural language and exploits the fact that each syntactic element has a designated semantic role – these semantic roles form the basis of expressive pointcut expressions in our RDL.

3.1 RDL Elements

The elements constituting our RDL are presented in the RDL metamodel in Fig. 3.

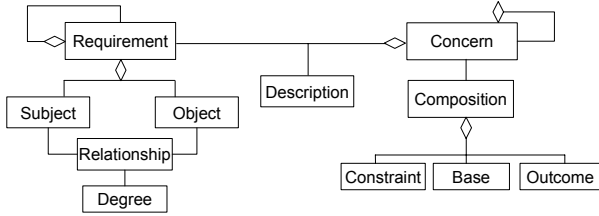


Figure 3: RDL metamodel

The RDL is an XML-based language – the XML annotations help automate the analysis (as discussed in Section 4). We discuss the various RDL elements and its composition mechanism next.

3.1.1 Description Elements and their Use

Our RDL is based on the symmetric view of AOSD [23, 24, 34, 35]: we use the same abstraction, i.e., a *Concern*, to represent both crosscutting and non-crosscutting elements. A Concern is a high-level unit for system partitioning, a container for localizing semantically related requirements (e.g., selling, account management, etc.). For instance, the *Reserve Room* concern in Fig. 4 localizes the details related to room reservation. The initial set of concerns for each system can be selected from a concern repository [24], or identified via mining tools, e.g., [5, 31], domain analysis, stakeholder interviews, or ethnographic studies.

A concern can be simple (containing only requirements), or composite (containing other concerns as well as requirements), thus allowing hierarchical structuring of related requirements. Each concern is identified by its name and encapsulated within `<Concern/>` tags.

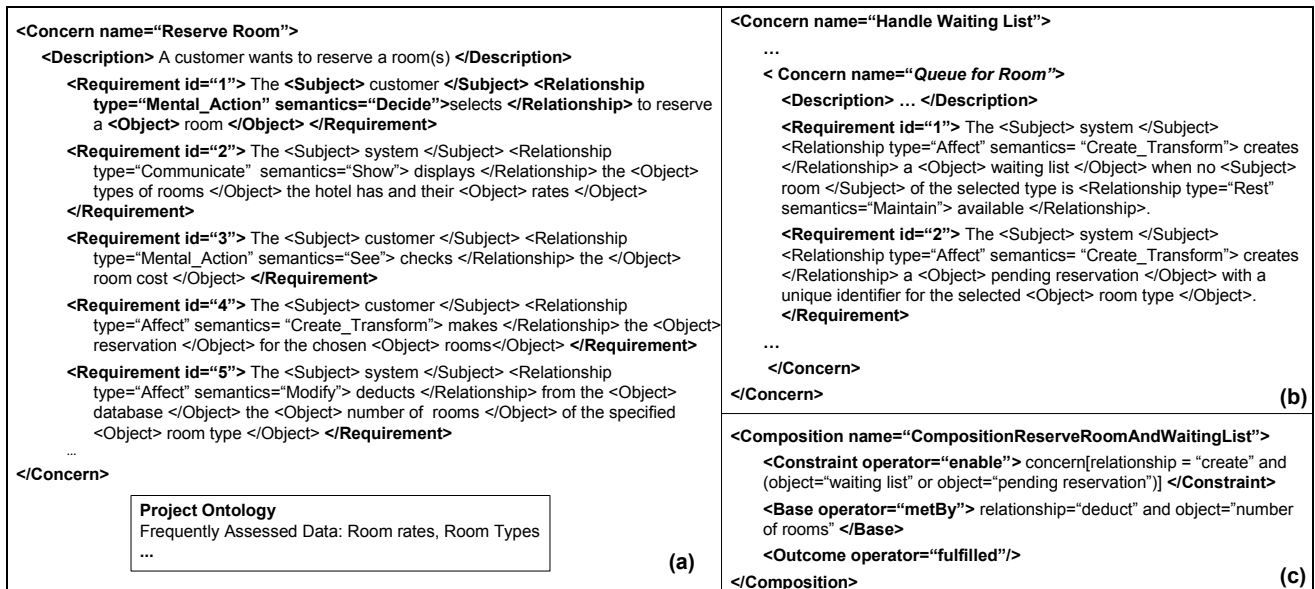


Figure 4: Example of RDL elements

A *Requirement* is a description of a service the stakeholders expect of the system, the system behavior, and constraints and standards that it should meet [33]. One or more requirement elements are encapsulated within a concern; each requirement is identified by a unique identifier² (unique within its defining scope, which is the concern). Similar to concerns, requirements too can have sub-requirements.

The meaningful concern names and requirement ids are neither necessary nor utilized in the RDL compositions. They are used for ease of referencing in discussion and are a part of RE tradition.

Both *concern* and *requirement* can be multi-sentence elements. However, the smallest unit of meaningful interaction conveying a construct is a simple sentence, and in order to be able to reason about this “smallest meaningful construct” we need to define

elements for its description. The main such elements in our RDL are *subject*, *object*, and *relationship*.

A *subject* is the entity that undertakes actions described within the sentence clause². Subject in our RDL corresponds to the grammatical subject in the clause. In order to support composition specifications involving a subject denoted with different words representing the same semantics, a set of synonymous definitions must be provided. These synonyms could be provided either through a readily available standard electronic synonyms dictionary (e.g., WordNet [2]) or per project, augmenting a standard dictionary with a project-specific ontology. Fig. 4(a) demonstrates a set of subjects marked within requirements (e.g., *customer* in requirement 1, *system* in requirement 2).

An *object* is the entity that is affected by the actions undertaken by the subject of the sentence, or in respect of which the actions are undertaken. Object in our RDL corresponds to the grammatical object in the clause. A clause could have several objects associated with (affected by) a single subject (e.g., in

² A requirement may contain one or more sentences. We do not need to number each sentence separately. A sentence may have one or more clauses.

requirement 2 of Fig. 4(a) both *type of rooms* and *rates* are objects.).

Relationship depicts the action performed (state expressed) by the subject on or with regards to its object(s). Relationships can be expressed by any of the verbs or verb phrases in natural language (e.g., *display* in requirement 2 of Fig. 4(a)).

The subject-relationship-object (S-R-O) structure carries the main semantic load of a sentence. Whereas subjects and objects denote the entities of significance in it, the relationship (i.e. verb) reflects the interaction between these entities. In our approach the relationship denotes the most central function, as it defines the functionality and/or properties that the subjects and objects provide. In order to be able to reason about the various types of relationships, we have used and adapted the linguistic studies of Dixon [12], Hale et. al. [14], and Levin [20] that classify the verbs in accordance with their semantics (interested reader is referred to Chitchyan et. al. [8, 9, 10] for more details). As a result of this (adapted) classification, we have a set of verb classes and sub-classes (depicted in Fig. 5) that cover all English language verbs³. Each such class has a set of common semantic roles and depicts their participation in a semantically related activity. For instance, all verbs of the Affect type involve three basic semantic roles – an Agent which moves or manipulates something (referred to as Manipulator role or Manip) so that it comes into contact with some thing or person that plays the Target role. Either the Manip or Target (or both) will be physically affected by this activity.

In Fig. 5 the text in italics represents the composition operators (discussed in the next section) derived from these relationship semantics.

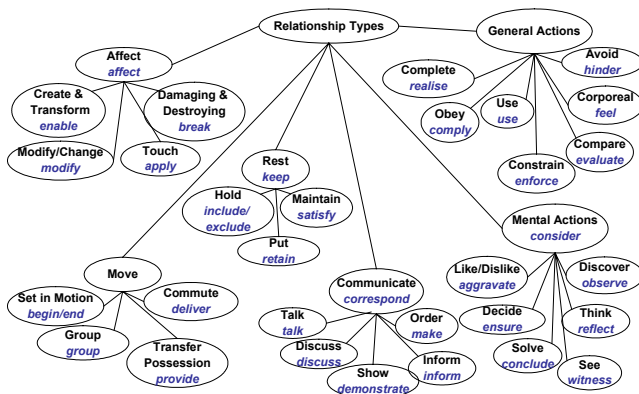


Figure 5: Verb classes and composition operators (shown in italics)

When specifying requirements stakeholders often qualify how important or significant a specific functionality or property is to them. In the RDL such qualifications are represented by the *Degree* element. Degree element depicts the strength of the relationship between the subject and object. An example for degree annotation is: `<Degree type= "Modal" semantics= "Want" level= "medium"> wants </Degree>` for the *wants* word

³ Though this particular work focuses on English, the principle of verb classification is independent of language [12]. The same approach may be applied to other languages and a similar (though not identical) classification will result.

in “A customer *wants* to reserve a room” *description* sentence from Fig. 4(a) – note that we omitted these annotations from Fig. 4(a) for simplification.

The set of degree categories currently used in the RDL, i.e. the *type* attribute of Degree element in the RDL (not shown in Fig. 4) are presented in Fig. 6. Each of these classes reflects a certain level of degree, for instance Maximizers (such as highly, completely, etc.) amplify some property to the top level, while Boosters (e.g., amply, considerably) amplify it but not to the maximum.

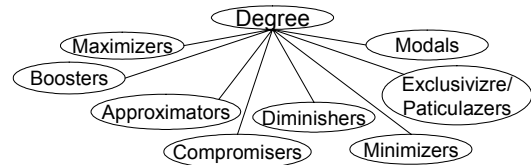


Figure 6: Degree classes

Of particular interest here is the group of Modals and related verbs: when a modal degree word (e.g., must, could, wish, etc.) qualifies a relationship, the relevance of that relationship (i.e. functionality or property) to the stakeholder is reflected (“must check” and “wish to reserve”). The Modals have related levels, which can be high, medium, or low, depending on the semantics of a particular verb.

The RDL elements above are used for requirements description, irrespective of their modular structuring or contents. Composition is discussed next.

3.1.2 Composition Elements and their Use

Composition is the assembling of the separately defined requirements modules with the aim of ensuring their desired interactions and revealing/addressing the undesired ones. A composition element in the RDL comprises three sub-elements: Constraint, Base and Outcome. An example of composition is presented in Fig. 4(c).

A *Constraint* element specifies what checks and restrictions are to be placed on a set of requirements (provided by *Base* element) and what action must be taken in imposing these constraints. The required actions are specified by the constraint *operators* (see elements in italics in Fig. 5) which are derived from the relationship categories. Since each verb group has a dedicated meaning and a set of related roles, the corresponding operator denotes that these roles are expected to participate in a specific interaction. Thus, for instance, the *enable* operator used in Fig. 4 (c) is derived from the *Create and Transform* subgroup of the *Affect* group, and so has the semantics of *Agent* manipulating the *Manip* in order to create something, i.e., the *Product* role. These roles are normally filled in by the elements picked out by the Constraint and Base queries.

The query specified within the Constraint element is used to select concerns/requirements that will act as the constraint to be imposed (cf. query expression enclosed in the `<Constraint/>` tags in Fig. 4(c)). The query is semantics-based, i.e. it selects elements by their meaning, rather than structure or id or name. The benefits of such semantic queries are twofold. Firstly, we avoid syntactic matching in the composition specifications, thus avoiding unintended element matching. Instead compositions are specified based on the semantics of the requirements – for instance, the constraint query in Fig. 4(c) selects “the concern in which a waiting list or a pending reservation is created” (more details are

provided in section 3.2.1). Secondly, it ensures that requirements compositions are semantically justified, rather than arbitrarily provided by a requirements analyst.

Base element provides a query for selecting the set of (points in) requirements that are affected by some constraints (provided by the *Constraint* element) and the temporal or conditional dependency between these requirements and the constraints. The query expression of the *Base* element, like that of the *Constraint*, is a semantic query. For instance, the base query in Fig. 4(c) selects “the requirement where number of rooms is deducted” (details are provided in section 3.2.1).

The temporal and conditional dependencies are depicted by the base *operators*, which are founded on sequencing and conditional semantics in natural language and reflect the ordering or conditional dependencies of requirements. Expressions such as “first... then”, “once”, “if” and alike are used to express such semantics. Our base operators fall into three categories:

- Sequential temporal operators: e.g., before, after, meets;
- Concurrent temporal operators: overlap, during, starts, finishes and concurrent;
- Conditional operators: if and if not.

The temporal operators (Fig. 7) fully describe the relative temporal positioning of one item with respect to another [4], and all, besides concurrency, can be symmetrically inverted into another operator (e.g. *before* can be inverted to *after*, etc.). Since in the example in Fig. 4(c) we want to specify that the *Queue for Room* concern should be enabled “immediately after” the number of rooms is deducted, we use the *met by* operator instead of the *after* operator which is applicable when a time lapse is involved. Note that *met by* in our RDL is equivalent to an “after” operator in languages such as *AspectJ* while *after* in our RDL is more analogous to deferred execution.

| | | |
|------------------------------|--|--------------|
| X before (after) Y | There is a temporal interval between requirements X and Y when X has completed but Y has not started yet. | XX YY |
| X meets (met by) Y | There is no temporal interval between requirement X ending and requirement Y starting (from perspective of X). | XXYY |
| X overlaps (overlapped by) Y | Requirement X has commenced before requirement Y; Y commences while X is in process; X completes while Y is in process (from perspective of X). | XXX YYY |
| X during (through) Y | Requirement Y has commenced before requirement X; X commences while Y is in process; X completes while Y is in process (from perspective of X). | XXX YYYYY |
| X starts (started by) Y | Requirement X has commences simultaneously with requirement Y; X completes while Y is in process (from perspective of X). This is a sub-type of during. | XXX YYYYY |
| X finishes (finished by) Y | Requirement Y has commenced before requirement X; X commences while Y is in progress; X and Y complete simultaneously (from perspective of X). This is a sub-type of during. | XXX YYYYY |
| X concurrent Y | Requirements X and Y are started and completed within the exact same temporal interval. | XXX YYY |

Figure 7: Selection of temporal operators for composition (based on Allen [4]). Here X and Y depict two requirements and their positioning towards each other reflects their temporal order.

The *Outcome* element defines how imposition of constraints upon the base sets of requirements should be treated. For instance, the outcome element may specify a set of requirements that must be *satisfied* as post-conditions upon application of the *Constraint* or merely state that the *Constraint* has to be *fulfilled* (as is the case in Fig. 4(c)). We will show an example of the use of *satisfied* in the *Outcome* element in Section 3.2.2.

We next discuss how our RDL facilitates expressive, semantics-based pointcut specifications.

3.2 Semantics-based Pointcut Expressions

Unlike previously discussed approaches, in which the smallest unit of reference for composition was a requirement, in RDL we use the elements that build the meaning of a requirement. Thus, we are able to select requirements by referencing their semantics without having to rely on any syntactic id.

To illustrate the use of the RDL for composition, we will first discuss how the examples presented in Section 2 (for the UCA and ARCADE approaches to AORE) can be represented in the RDL. Then we will discuss some additional uses of the RDL.

3.2.1 Queue for Room Composition

The room booking example, discussed in section 2.1, is structured into use cases and use case steps. The same modularization structure in our RDL is illustrated in Figs. 4(a) and 4(b).

The composition for the Reserve Room and Queue for Room concerns is shown in Fig. 4(c). The *Constraint* query of this composition states that those *concern(s)* should be selected (the square brackets) that contain one or more requirements that involve *creating* (relationship) a *waiting list* (object) or a *pending reservation* (object). This query will match the *Queue for Room* concern in Fig. 4(b), as the relationship and objects of its requirements with ids 1 and 2 correspond to the *Constraint* query criteria. Thus, the *Queue for Room* concern will be selected without any reference to its name or to any of the ids of its requirements. If a synonym of the *create* verb were used in the query instead, the respective requirements would still be matched. The *Base* query of this composition refers to requirements (as there is no “concern []” expression used) that involve *deducting* (relationship) *number of rooms* (objects), which matches the requirement with id=5 in Fig. 4 (a). Again, the point of interest for composition is defined without any syntactic reference. The *Base* operator is *met by*, meaning that the constraint should be imposed immediately after the base.

Thus, the composition in Fig. 4(c) states that *immediately after* each requirement where the system “*deducts the number of rooms*” the concern containing requirements with system “*creating a waiting list or a pending reservation*” must be enabled. This intention of the composition is clear from the composition specification itself, without the need to look up anything from the participant concerns, thus providing a single point of reference for compositional reasoning.

This composition specification is also robust, unlike those in the previously discussed fragile syntactic counterparts. As long as queuing for room needs waiting lists or pending reservation creation, the constraint of the composition will not change. Similarly, as long as the check for queuing is to be applied upon room number deduction, the base also will be unaffected. This is true irrespective of any number of new requirements and concerns being added or removed from the specification.

The semantic queries used in the above example demonstrate that we can use the subject-relationship-object structure to find both the larger-grain entities, such as concerns, or requirements within which the given structure occurs, as well as a particular point, i.e. part of the sentence which contains this structure. In our automation of the RDL, if no “concern []” expression is used, we refer to the S-R-O encapsulating requirement, though more fine-grained interpretations are also possible.

3.2.2 Cache Access Composition

The second composition of the room booking example in section 2.1 is related to composition of the cache access use case. The UCA approach was forced to define extension points within each application use case and match the Perform Transaction template to each of these use cases. The composition definition for this case in the RDL is presented in Fig. 8.

The constraint query of this composition selects the concern(s) (concern [] statement in Constraint) where *look up* (Constraint relationship) of *cache* (Constraint object) takes place. The base operator is *meets* implying that the constraint must be applied immediately before the base (this is the same as “before” in AspectJ⁴). Finally, the Base query selects any requirements where “frequently used data” (subject or object in Base) performs some action (i.e., is a subject) or is affected (i.e., is an object). Thus, the composition states: “apply concern where system looks up cache immediately before any requirement where ‘frequently accessed data’ appears”. In our approach we need only to complement this query with an entry defining “frequently assessed data” into the project ontology. No external joinpoint definition, id or name matching within the composition is needed. The composition intention is also completely obvious from the statement itself.

```
<Composition name="CompositionAccessCache">
  <Constraint operator="apply"> concern [relationship=
    "look up" object="cache"] </Constraint>
  <Base operator="meets"> subject="frequently used data"
    or object=" frequently used data" </Base>
  <Outcome operator="satisfied">relationship ="update" and
    object="cache" </Outcome>
</Concern>
```

Figure 8: Composition Access Cache in the RDL

Moreover, having used the Outcome element, we also ensure that any requirement to update cache is also fulfilled as a result of this composition.

It is worth noting that the same composition shown in Fig. 8 for Cache Access, written for the use cases structure, will satisfy the composition definition of the concerns from the ARCADE example discussed in section 2.2. This is true because though these different approaches had structured the requirements into different models, the semantics of these requirements and their dependencies have remained unchanged. Consequently, the composition specified on the basis of these semantics and dependencies in our RDL is applicable regardless of the requirements structuring approach.

3.2.3 Assignment Operators in Composition

It should be noted that the semantic match for the queries defined by the S-R-O structure can be realized either per word *lemma* only, or per word *synonyms*.

A *lemma* is a reduction of the surface forms of the given word to its corresponding dictionary headword. For instance, *systems* will be reduced to *system*; *looking* to *look*, etc. When matching “look up” per lemma other surface forms of it (e.g., looking up) will also be matched, but no other synonym will.

On the other hand, when matching by synonym, both lemmas and synonyms of the given word are matched, so for instance, in case of “look up”, not only *looking up*, but also *search for*, *find*, *hunt for*, etc. will be matched.

Both of these matching techniques are significantly more expressive than the per string syntactic matching.

Normally the synonym-based matching is used in the RDL when the “=” operator is used. However, if required, a per lemma matching is also available via use of “= ” operator. This allows to define more narrow queries when needed, as well as to avoid confusion when a word is used in a specific sense but no project-specific lexicon is provided. For instance, with a standard synonym dictionary, if used with synonym assignment, *system* will match such words as *scheme*, *coordination*, *organization*, etc. However, we want only to match *system* in the sense of the computer system to be developed, so, if we have no project specific synonym dictionary or ontology of terms, the narrow per lemma assignment will be used.

3.2.4 Verb Classes in Composition

In addition to using the semantics of the S-R-O structure, other semantics of the RDL elements can also be used for querying. For instance, the semantics of the verb classes and sub-classes (shown in Fig. 5) can be used to select requirements related to broad types of activities. One such example is shown in Fig. 9.

```
<Composition name="CompositionAccessCache">
  <Constraint operator="apply">concern [
    relationship.semantics="Modify" and object="cache"]
  </Constraint>
  <Base operator="metBy"> relationship.semantics=
    "Create_Transform" and object="frequently used data"</Base>
</Concern>
```

Figure 9: Example of composition with verb class semantics

In this composition the constraint query is very similar to that of Fig. 8, but the *update* relationship is described in terms of its semantic category: “Modify” (*semantics* attribute of the Relationship element in RDL used to capture the sub-class of a generic verb class to which a given verb belongs (Fig. 5) [10]⁴), thus the constraint will select the concern that contains requirements where the cache is modified.

Similarly, the relationship in the Base query refers to the semantics attribute of the Relationship element to select all instances where the frequently used data (object) occurs with a verb of class *Create_Transform*, which includes such verbs as, for instance, create, make, generate, produce, form, build, construct, forge, mix, stir, cook, etc. While some of these verbs are unlikely to be used with the given object, the others will depict a certain type of interaction reflected by the *Create_Transform* sub-class: i.e. the Agent role (e.g., system) manipulating (e.g., create) the optional Manip role (e.g., records) into the Product role (e.g., pending reservation).

Such broader-view queries are particularly useful when a requirements analyst wishes to establish an understanding of a particular interaction. Realistically it is not possible to write such queries either in UCA or in ARCADE composition, as in both cases all individual verb statements in all concerns and requirements will have to be checked for the specific verbs and their corresponding extension points created/syntactic ids found and then listed in a pointcut/composition expression.

⁴ The full schema for the RDL and meaning of each attribute is available from [10].

3.2.5 Degree Element in Composition

The Degree element and its attributes can also be used in queries. A query by relationship-related degree is shown in Fig. 10. In this example the constraint query does not select the constraint from the requirements, but provides it within the query statement.

The Base query selects requirements where the degree element of Modal type is used, as shown by the *degree.semantics*= "Modal" part of the query. The Modal type, due to the nature of the items included in this group, will always qualify verb phrases, i.e. relationships. This type of degree elements shows the level of the pertinence of the action/property defined by the verb phrase. The part of query stating *degree.level*= "high" uses the *level* attribute of the degree element which can be given high, medium, or low levels [10]. Thus, the base query will select the requirements where the relationship has a high level of pertinence, which may include requirements like "The credit card *must be validated* for payment", etc.

```
<Composition name="CompositionFirstRelease">
  <Constraint operator="include"> "Include requirements into
  1st release" </Constraint>
  <Base operator="if"> degree.semantics="Modal" and
  degree.level="high"</Base>
</Concern>
```

Figure 10: Example of degree element in composition

In addition, other types of the degree element (Fig. 6) indicate how the quality to which the degree relates should be treated. For instance, *degree.semantics*= "Maximizer" will select the set of requirements where a certain property must be maximized up to some given top level, while in case of use of Diminisher type the given level should be reduced, if possible.

4. Automation Support for RDL

Above we have presented our RDL and discussed the expressiveness and simplicity of composition specification that it allows. However, it is clear that none of these benefits will be adequate if one has to manually annotate the requirements text with the RDL or manually match the composition specification queries to the annotated requirements. For this reason we briefly discuss the automation support that we have implemented for the RDL. The annotations are automated through an extension of our existing Wmatrix natural language processing tool suite [30] while the analysis (based on the semantics-based compositions) is automated by the MRAT Eclipse Plugin [36].

4.1 Automating Semantic Annotation

Automation for the RDL annotations is provided through an extension of Wmatrix, a web-based suite of corpus-based natural language processing (NLP) tools [30]. Existing annotations in Wmatrix include part-of-speech (POS) and semantic tags. Wmatrix also incorporates frequency profiling, keywords and concordancing (i.e. words in context) retrieval tools. Keywords analysis identifies concepts of potential significance in a given text. Wmatrix assigns POS tags to each word in the text with 98% accuracy using a hybrid statistical and rule-based method. The underlying semantic annotation system initially assigns a tag to each word in the text from a general-purpose taxonomy [29].

The accuracy of the NLP components within Wmatrix has been evaluated in terms of a number of criteria (e.g., precision, coverage and multiword expression recall) and over a range of domains and document styles. The robustness of statistical language models embodied in Wmatrix has been evaluated on requirements and standards documentation (highly technical genres) through to ethnographic field reports (a written genre that is much closer to spoken language). These are reported by Sawyer et al. in [38].

In order to support text annotation for the RDL described in this paper, we have extended the functionality of Wmatrix in three areas by:

- marking major grammatical constituents of sentences such as subject, verb and object;
- explicit output of *lemmatization*, i.e. reducing similar words such as *systems* and *system* to the same base word (dictionary headword);
- classification of verbs.

4.1.1 Subject, Verb, Object Identification

In order to mark the grammatical subject, verb and object in each sentence, we build a set of links on top of the tags already assigned by the POS tagger. The links are inserted by matching flexible patterns of POS tags. These patterns have been identified by linguists using a combination of linguistic knowledge and corpus evidence. We have adapted a set of existing rules for marking grammatical relationships in the SketchEngine system [18] used in the dictionary publishing industry. For example, a simple rule to link a verb to its object is as follows:

```
N*o[.] (RR*/RG*/XXn3) VVN*v[.]
```

This matches the sequence 'Noun' (N*), followed by between 0 and 3 possibly negated 'adverbs' (RR*/RG*/XX), followed by a past participle 'verb' (VVN). In the case of a match, the noun is marked as the object of the verb. The subjects, verbs and objects are marked explicitly by Wmatrix along with the result of lemmatization.

4.1.2 Verb Classification

The verb classification discussed in section 3.1 (Fig. 5) is based on the work by Dixon [12] and is not completely aligned with the semantic classes of words used by Wmatrix [30]. In order to support this classification in Wmatrix we have defined a new tagset for verbs based on the RDL classification and have established a mapping from the Wmatrix verb categories onto this new tagset. In some cases the large classes of Wmatrix words were directly compatible with the RDL verb classification, for instance, the verbs of domain for *Movement*, *Location*, *Travel and Transport* in Wmatrix largely correspond to the RDL verbs of Motion type. On the other hand, there are semantic classes in Wmatrix (e.g., Education, Time, etc.) tagset which have no correspondence to that of the RDL verbs tagset and their contents have been mapped to the RDL verb classes on individual verb by verb basis. For instance the *graduate* verb from the Education class was mapped to the *Commute* sub-type of *Move* type of the RDL classification, while the teach verb from Education was mapped to *Think* sub-type of *Mental Action* type for RDL classification.

4.1.3 Use of Wmatrix Classification

While the verb classification for RDL was significantly different from that of Wmatrix, the classes of other RDL elements were

much closer to the existing tagsets of Wmatrix. So, for instance, the Degree element of the RDL directly reuses the degree classes of Wmatrix (e.g., Maximizers, Minimizers, Approximators, etc.), introducing only some additions, mostly related to the use of modals and secondary type verbs (i.e. verbs that modify the meaning of other verbs, e.g., wish, hope, etc.) for degree reflection.

4.2 Automating Composition and Analysis

The RDL annotations from Wmatrix are used for requirements analysis in the Multidimensional Requirements Analysis Tool (MRAT) [36]. MRAT focuses on analyzing the temporal relationships of RDL compositions. Finding these temporal dependencies is primarily useful for determining the points of sequencing conflicts and points that, by virtue of commonality, are potentially affected by problems. Temporal conflicts and inconsistencies can arise since each composition specification may define a temporal sequence for its elements and the same elements may participate in different compositions with incompatible operators. In addition, recording of base operator relationships aids requirements understanding by showing the relative positioning and dependencies of requirement elements.

The composition algorithms of MRAT reason about the temporal relationship of base and constraint requirements in composition specifications. Since the RDL does not restrict participation of any element in any number of compositions, potentially, every requirement in one base/constraint pointcut expression can be found in another composition, or even elsewhere in the same composition. Thus, potentially every requirement should be tested against every other requirement for temporal interactions. To reduce the number of considered interactions to the most pertinent ones, MRAT applies the notion of *compositional intersections* from Moreira et. al. [24], i.e. analyzing only the requirements that are in common for more than one base or constraint element.

MRAT compositional intersections are built by evaluating, against each other, combinations of requirements and concerns selected by composition queries. A mathematical intersection of these requirements for each two interacting base or constraint elements is formed. If this set is non-empty then it is compared against existing compositional elements to find equivalent sets. If no equal set is found a new intersection is created. Hence, an MRAT compositional intersection may contain multiple pairs of base and constraint elements from which it is derived. This algorithm is sketched in Fig. 11, where Fig. 11(a) shows the flow of the main steps, while Fig. 11(b) depicts intersection formation between two compositions [36]).

Having found the points of possible pair-wise interactions between requirements, MRAT can now analyze these for problems. Since all pairs of requirements that interact are recorded by intersections, analyzing these relationships and resolving them will ensure that the complete specification is problem free.

To facilitate such analysis, we build graphs for relationships between requirements. To build the temporal graph for requirements, the base to constraint temporal relationships, originating from the temporal composition operators in the Base elements of respective compositions, are recorded as links for each intersection. These links are represented in multiple base-operator graphs. The links are denoted as *specified* (i.e. directly taken from the type of temporal operator in composition

specification) and their *inverse* (i.e. the symmetric opposite of specified, e.g., the inverse of *before* is *after*). The requirement graphing algorithm additionally denotes link types as *transitive* and its inverse (called *transitive inverse*). *Transitive* links exist between two requirement elements that are related by virtue of both linking to a common third requirement (e.g., if X is *before* Y and Z is *after* Y then the transitive link for X and Z can be formed as *X before Z*).

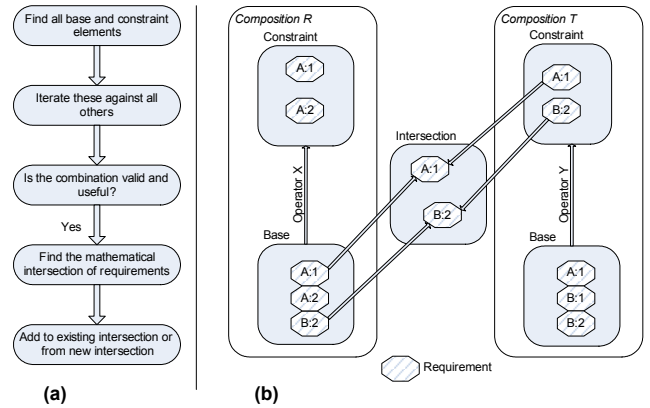


Figure 11: Compositional intersection derivation in MRAT

Such link derivation is a complex task due to the inherent ambiguity of combining temporal operators (shown in Fig. 7) to a single resultant operator, though intuitively we know that some operators are compatible, while others are not. For instance, if we have (X *before* Y) and (Y *meets* Z), this implies that X is before Z. However, from (X *starts* Y) and (Y *overlaps* Z) no obvious link can be created between X and Z as it is not clear how they are related: will X complete and then Z commence, or will X still be in progress at the same time with Z? Thus, to cope with this ambiguity a certain operator order has been established and a mapping function for transitive link operator derivation has been developed [36]. The operator ordering is depicted in Fig. 12.

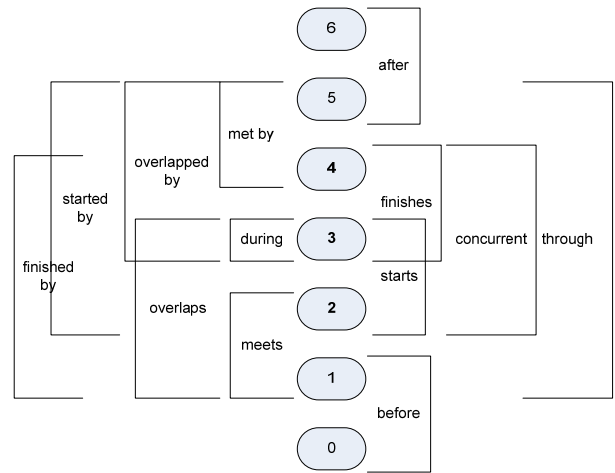


Figure 12: Temporal ordering of operators in MRAT [36]

The ordering decisions are based on the notion of operator splitting: it is assumed that each operator can be split into a combination of the 7 points (0-6) depicted in Fig. 12. For a requirement that participates in two compositions both

composition operators are mapped to the presented stages. A *redefinition* is identified when at least one of the points of the first operator is one unit away from one of the points of the second operator. For instance, comparing *before* and *meets* operators leads to a redefinition since 2 is 1 unit away from 1. This is intuitively understandable, as if X is before Y and X meets Z we know that X is also before Z, though it is much closer to Z than to Y. If the difference between operator stages is 2 or more, a conflict is identified, as requirements cannot in a compatible way satisfy both temporal operators.

Thus, conflicts, redefinitions and requirements potentially affected by them are found by scanning the requirement graph. The identified problems are then presented on the UI by tree items, visualization and contextual build messages in the source editor. A screenshot for part of this UI is depicted in Fig. 13. Here the vertical lines represent the time-lines which inversely correspond⁵ to those of Fig. 12. The visualization is presented with respect to a requirement selected in the left-hand tree (not shown in Fig. 13). A set of different visual links are used to depict redefinitions and conflicts. Thus, the users may explore the compositions selecting requirements and examining the temporal ordering of all other affected concerns/requirements and their conflicts and redefinitions.

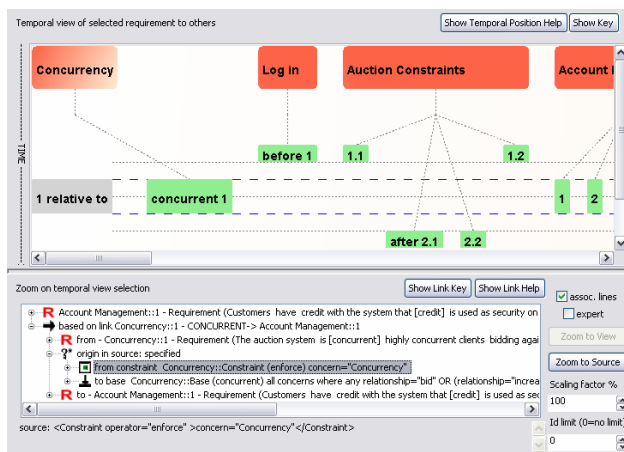


Figure 13: A visualization for temporal ordering of requirements in the MRAT user interface

5. Discussion

In the introduction, we noted that the syntactic references to requirements (e.g., direct references to requirement identifiers, or named points) cause four problems for the requirements composition. In this section we discuss how our RDL and its implementation address these four problems.

The problem of *loss of intentionality* is addressed by our RDL defining compositions directly in terms of requirements semantics. As shown in section 3.2, the RDL compositions are expressed in terms of the entities (i.e., subjects and objects), their interactions (i.e., relationships), properties (i.e., degree), and ordering. This supports compositional reasoning as the requirements engineers' intention is clearly captured in semantic

terms rather than through references to numeric ids or labels. This is complemented by the visualization of the temporal view of composition in MRAT, which further helps reason about requirements interdependencies and relationships as captured by the semantics-based composition specifications.

The problem of *individual joinpoint definition* is completely eliminated by the Wmatrix automated annotation⁶ of the RDL and use of synonym dictionaries and project-specific ontology. Each annotated element becomes a potential joinpoint shadow, and can be referenced via its grammatical function (e.g., subject, object, relationship), semantic type and lemma or synonym in the compositions. Note that the semantic queries in the RDL compositions will always match all requirements/concerns with the given semantics. This provides strong semantic underpinnings for trade-off analysis as well as supports duplicate requirement removal and crosscutting concern identification. For instance, one may write a query to find all requirements which include a relationship of type *Affect* with *Damage* semantics (i.e., a verb from the *Damaging and Destroying* sub-category of the *Affect* relationship category. This could help identify a crosscutting concern specifying requirements for harmful situations.

The issue of *trade-off analysis* requires support for identifying and resolving problems potentially caused by interacting concerns. Effective identification of such trade-offs is essential for being able to reason about the global properties of the system. The trade-off points revealed by MRAT based on the semantic queries in the RDL composition specification do not just reveal shared join points but also expose the more subtle interactions among aspects affecting the join points. For instance, if aspect *A* uses the *modify* operator in its constraint query while aspect *B*, applying at the same join point, uses the *hinder* operator then, using the semantics of the composition operators, the requirements engineer can clearly reason that there is a conflict, i.e., *B* would hinder the modification that *A* wishes to impose. On the other hand, if *B* were employing the *use* operator then the engineer can reason with confidence that it uses the requirement at the join point and does not stop *A* from modifying it. Of course, it then remains up to the requirements engineer to decide whether *A*'s modification should precede *B*'s use or vice versa. This is facilitated by the temporal ordering view in MRAT, which can provide a default ordering based on the temporal ordering of operators described in Fig. 12. The final decision on the resolution, however, cannot, and in fact must not, be automated as it is best left to the requirements engineer's or domain engineer's understanding of the problem.

The problem of *pointcut fragility* does not arise in the RDL-based pointcut specifications as they contain no structure-related references, except for the S-R-O structure which carries the actual semantics of a requirement. This was demonstrated in section 3.2.2 where we noted that the same RDL composition statement is applicable for both use case-style and ARCADE-style structuring of the requirements. Furthermore, since the composition specification refers to the semantics of the natural language used to specify the requirements, if new requirements with matching semantics are added we can be confident that they will be matched by the queries. In a syntactic composition model one cannot be sure how effectively a pointcut expression will match

⁵ This is to say that time flow is shown from bottom up in Fig. 12 and from top down in Fig. 13.

⁶ Other Wmatrix annotations, e.g., POS and statistical details, are utilised for requirements structuring and aspect identification [31].

relevant new labels and ids. Furthermore, if the semantics of a requirement change or it is removed, it will no longer be matched by the composition queries, hence avoiding semantically unintended matches.

One could argue, that some of the problems discussed above originating from syntactic referencing could be resolved if meaningful naming and tool support were provided. This may be true if consistent cross-references were maintained and textual descriptions linked to ids. However, more serious problems of syntactic pointcuts, like unintended matching (e.g., with * or “all”), necessary extension enumeration, and pointcut fragility will persist even with such tool support and meaningful naming. On the other hand, our RDL moves away from the pointcut specification by extension (i.e., by enumeration) to that by intention (i.e., semantics of the pointcut).

It is worth noting that our semantics-based composition specifications could be reused in isolation and also as part of complete requirements collaborations. A case of composition specification reuse was demonstrated in section 3.2, where we discussed how the same composition (from Fig.8) was used for both use-cases and Arcade-based requirements structures. Reuse of compositions as part of requirements collaborations will arise when the domain knowledge modeled via requirements and their interactions is reused. The only condition required for correct reuse of compositions is that the synonym vocabulary or ontology used for the composition specification should also be used along with the composition, and if necessary, updated to accommodate the (change of) project-specific terminology.

6. Related Work

The issue of expressiveness of composition mechanisms, and in particular of pointcut languages, has been widely discussed within the AO language engineering community. For instance, Ostermann et. al. [25] presents a pointcut language which utilizes a combination of 4 different models of program semantics (the AST, the execution trace, the heap, and the static type assignment), thus exposing more semantic information than any one model can provide. This richer information and the focus on “what” to compose (rather than “where” or “how”) is a significant step towards improved expressiveness. Lopes et. al. [21] discuss the lack of expressiveness of contemporary programming languages and point to the rich referencing mechanisms of natural language as the vein for improved expressiveness. Knoll and Mezini [19] outline a “naturalistic” programming language Pegasus to bring programs closer to the natural language, and thus, utilize the expressive binding mechanisms of natural language.

Nevertheless, these issues are yet to be raised in the Early Aspects area. As discussed in section 2, the dominant approach to pointcut definition in AORE today is syntactic matching: either via individually naming the joinpoints and referencing them in pointcuts, as was discussed for UCA [16], or by using ids as joinpoints as done in ARCADE and its follow up work [24, 28]. The same approach to naming (e.g., by listing in dedicated sections of requirements templates) or assigning ids to requirements is also used by Brito and Moreira [6], Moreira and Araujo [22], Whittle and Araujo [37] and other works many of which are available from the Early Aspects portal [3].

Some similarity to our work, in using the natural language (NL) processing and lexical analysis, can be found in EA-Miner [31],

Theme/Doc [5, 11], and CCVerbFinder [32] tools. EA-Miner supports crosscutting concerns identification in NL requirements by matching these requirements against a lexicon dedicated to known types of the generally crosscutting concerns (e.g., security, persistence, etc.). Theme/Doc uses lexical analysis for identification of interdependent activities (i.e., verbs) in NL requirements, as well as structuring verbs into a set of synonyms for theme construction. The crosscutting concern lexicon in EA-Miner and the verbs in Theme/Doc may act as types of semantic joinpoints, though neither EA-Miner nor Theme/Doc tools actually support pointcut definition or composition. CCVerbFinder, on the other hand, uses NL processing for action verb-based feature identification in code rather than requirements, though it is interesting to note that it too uses the notion of verb and object.

7. Conclusion

In the “Mythical Man-Months” [7] F. Brooks writes “...the essential difficulties are inherent in the conceptual complexity of the software functions to be designed and built at any time, by any method...”, i.e. software has an *inherent* complexity, however not all complexity in software development is inherent: the method that we choose to design and built the software often comes with its own, so called, *accidental* complexity.

As AOSD moves ahead with promoting modular and compositional reasoning, it is essential to eliminate elements of accidental complexity in its methods. The syntactic composition models in AOSD techniques in general, and AORE techniques that are the focus of this paper, aggravate accidental complexity. In contrast, our RDL abstracts away from these syntactic details hence helping the requirements engineer to focus on the inherent complexity of the problem. The semantics exposed by our RDL and the semantics-based queries used in the composition specification facilitate reasoning about the stakeholders’ intentions embedded within the natural language descriptions of requirements. Furthermore, the semantics-based approach in our RDL supports development of reasoning frameworks focused on the requirements semantics. This is demonstrated by the analysis of concern interactions in terms of constraint and base operator semantics in the MRAT tool. Our RDL and its supporting tools are, therefore, a stepping stone towards more expressive pointcuts and intentional reasoning about concern dependencies and interactions at the requirements-level.

Acknowledgments: The authors thank Adam Kilgarriff for providing SketchEngine rules for adaptation in Wmatrix, Prof. Robert France for discussions about software complexity, Prof. Andrew P. Black and the anonymous reviewers for AOSD’07 for their helpful comments on the previous version of this paper.

This work is supported by EC Grant AOSD-Europe: European Network of Excellence on AOSD (IST-2-004349) and EPSRC Grant EP/C003330/1: Multidimensional Analysis of Requirements Level Trade-Offs (MULDRE).

8. References

- [1] AspectJ Project, <http://www.eclipse.org/aspectj/>, 2006
- [2] WordNet, <http://wordnet.princeton.edu/>, 2006.
- [3] Early Aspects. net, <http://www.early-aspects.net/>, 2006.

- [4] J. F. Allen, "Maintaining Knowledge about Temporal Intervals", *CACM*, Vol. 26, No. 11, pp. 832-843, 1983.
- [5] E. Baniassad, S. Clarke, "Theme: An Approach for Aspect-Oriented Analysis and Design", *Int'l Conf. on Software Engineering (ICSE) 2004*, pp. 158–167.
- [6] I. S. Brito, A. Moreira, "Advanced Separation of Concerns for Requirements Engineering", *VIII Jornadas Ingeniería del Software y Bases de Datos (JISBD)*, 2003.
- [7] F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*. Reading: Addison-Wesley, 1995.
- [8] R. Chitchyan, S. S. Khan, and A. Rashid, "Modelling and Tracing Composition Semantics in Requirements", *Early Aspects 2006: Traceability of Aspects in the Early Life Cycle Workshop (held at AOSD'06)*, Bonn, Germany, 2006.
- [9] R. Chitchyan, A. Rashid, "Tracing Requirements Interdependency Semantics", *Early Aspects 2006: Traceability of Aspects in the Early Life Cycle Workshop Early Aspects WS (held at AOSD'06)*, Bonn, Germany 2006.
- [10] R. Chitchyan, A. Sampaio, A. Rashid, P. Sawyer, S. S. Khan, "Initial Version of Aspect-Oriented Requirements Engineering Model", *Lancaster AOSD-Europe report (D36): AOSD-Europe-ULANC-17*, 2006.
- [11] S. Clarke, E. Baniassad, *Aspect-Oriented Analysis and Design: the Theme Approach*: Addison-Wesley, 2005.
- [12] R. M. W. Dixon, *A Semantic Approach to English Grammar*, 2 ed. Oxford: Oxford University Press, 2005.
- [13] R. Filman, D. Friedman, "Aspect-Oriented Programming is Quantification and Obliviousness", *OOPSLA WS on Advanced Separation of Concerns*, 2000.
- [14] K. L. Hale, S. J. Keyser, "A View from the Middle", MIT, Center for Cognitive Science 1987.
- [15] I. Jacobson, M. Chirsterson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.
- [16] I. Jacobson, P.-W. Ng, *Aspect-Oriented Software Development with Use Cases*: Addison Wesley, 2005.
- [17] A. Kellens, K. Mens, J. Brichau, and K. Gybels, "Managing the Evolution of Aspect-Oriented Software with Model-based Pointcuts", *European Conf. on Object-Oriented Programming (ECOOP) 2006*, Springer Verlag, LNCS 4067, pp. 501-525.
- [18] SketchEngine, www.sketchengine.co.uk, 2006.
- [19] R. Knöll, M. Mezini, "Pegasus: First Steps Toward a Naturalistic Programming Language", *OOPSLA, Onward Track*, 2006, ACM.
- [20] B. Levin, *English Verb Classes and Alternations: a Preliminary Investigation*. Univ. of Chicago Press, 1993.
- [21] C. V. Lopes, P. Dourish, D. H. Lorenz, and K. J. Lieberherr, "Beyond AOP: Towards Naturalistic Programming", *Int'l Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) 2003*, ACM, pp. 198-207.
- [22] A. Moreira, J. Araujo, "Handling Unanticipated Requirements Change with Aspects", *SEKE 2004*, pp. 411-415.
- [23] A. Moreira, J. Araujo, and A. Rashid, "A Concern-Oriented Requirements Engineering Model", *Proc. Int'l Conference on Advanced Information Systems Engineering (CAiSE) 2005*, LNCS, Vol. 3520, pp 293-308.
- [24] A. Moreira, J. Araujo, and A. Rashid, "Multi-Dimensional Separation of Concerns in Requirements Engineering", *Int'l Conf. on Requirements Engineering (RE) 2005*, IEEE CS, pp. 285-296.
- [25] K. Ostermann, M. Mezini, and C. Bockisch, "Expressive Pointcuts for Increased Modularity", *European Conf. on Object-Oriented Programming (ECOOP) 2005*, Springer, LNCS 3586, pp. 214-240.
- [26] M. K. R. Kazman, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, "The Architecture Tradeoff Analysis Method", *Int'l Conf. on Eng. Complex Comp. Systems (ICECCS)*, 1998, IEEE CS Press, pp. 68-78.
- [27] A. Rashid, A. Moreira, "Domain Models are NOT Aspect Free", *MoDELS 2006*, Springer, LNCS 4199, pp. 155-169.
- [28] A. Rashid, A. Moreira, J. Araujo, "Modularisation and Composition of Aspectual Requirements", *AOSD 2003*, ACM, pp. 11-20.
- [29] P. Rayson, "UCREL Semantic Analysis System (USAS)". <http://www.comp.lancs.ac.uk/ucrel/usas/>, 2006.
- [30] P. Rayson, Wmatrix, <http://www.comp.lancs.ac.uk/ucrel/wmatrix/>, 2006.
- [31] A. Sampaio, R. Chitchyan, A. Rashid, and P. Rayson, "EA-Miner: a Tool for Automating Aspect-Oriented Requirements Identification", *Int'l Conf. on Automated Software Engineering (ASE) 2005*, pp. 352-355.
- [32] D. Shepherd, L. Pollock, and k. Vijay-Shanker, "Towards Supporting On-Demand Virtual Remodularization Using Program Graphs", *AOSD 2006*, ACM, pp. 3-14.
- [33] I. Sommerville, *Software Engineering*, 7ed: Addison-Wesley, 2004.
- [34] S. Sutton, I. Rouvellou, "Modeling of Software Concerns in Cosmos", *Int'l Conf. on Aspect-Oriented Software Development (AOSD)*, 2002, pp. 127-133.
- [35] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. Sutton, "N Degrees of Separation: Multi-Dimensional Separation of Concerns", *ICSE 1999*, ACM, pp. 107-119.
- [36] R. W. Waters, "MRAT: A Multidimensional Requirements Analysis Tool", *MSc. Dissertation*, Lancaster Univ., UK, Oct. 2006.
- [37] J. Whittle, J. Araujo, "Scenario Modeling with Aspects", *IEE Proc. – Software*, Vol. 151, No. 4, pp. 157-172, 2004.
- [38] P. Sawyer, P. Rayson, K. Cosh, "Shallow Knowledge as an Aid to Deep Understanding in Early Phase Requirements Engineering". *IEEE Trans. Software Eng.*, 31(11), pp. 969-981, 2005