Multi-Dimensional Composition by Objective in Aspect-Oriented Requirements Analysis

André Marques, Ana Moreira, João Araújo
CITI/Departamento de Informática
FCT, Universidade Nova de Lisboa
2829-516 Caparica, Portugal
+351-21-2948536

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

ABSTRACT

This paper focuses on composition of requirements artefacts in Aspect-Oriented Requirements Engineering (AORE). Our goal is to equip the Aspect-Oriented Requirements Analysis (AORA) approach [1, 2] with an enhanced composition mechanism. The AORA approach consists in the identification, modularization and composition of crosscutting concerns. But the AORA composition [3] operates at a coarser granularity level and its reduced number of operators results in limited composition expressiveness. The core of the work presented in this paper is the description of a composition language and approach that enables a multi-dimensional composition of artefacts based on *objectives*, therefore stating the purpose or the goal of the composition. This approach provides a composition process with a well-defined syntax and semantics, as well as a tool support integrated to the AORA original tool [3, 4].

Categories and Subject Descriptors

D.2.1 **[Requirements/Specifications]**: *Languages, tools*; D.2.2 **[Design Tools and Techniques]**: *Computer-aided software engineering (CASE)*; D.3.1 **[Formal Definitions and Theory]**: *Syntax.*

General Terms

Documentation, Languages, Management, Standardization.

Keywords

Early-Aspects, Aspect-Oriented Requirements Analysis, Multi-Dimensional Separation of Concerns, Composition, Objectives.

1. INTRODUCTION

The Requirements Engineering (RE) process is defined as a set of structured activities, to derive, validate and manage requirements documents of a given system [5, 6]. Aspect-Oriented Requirements Engineering (AORE) supports the traditional RE process by identifying crosscutting concerns and analysing its influences within the system. Many AORE approaches have appeared such as [1, 7-10].

The Aspect-Oriented Requirements Analysis (AORA) approach [1, 2] is a symmetric approach that includes the identification, modularization and composition of crosscutting concerns. However, the AORA composition rules proposed in [3] are too coarse grained and offer only few operators [11]. These limitations have a negative impact when moving towards later stages of software development, such as software architecture and design phases, when more detailed models must be produced. To solve this issue, we propose a multi-dimensional composition of concerns, be they functional or non-functional, where the driver of the composition is its objective. The ComBO composition language was created and a supporting tool developed and integrated in the existing AORA tool [3, 4]. To help the use of ComBO, a process was also elaborated to be integrated to the last step of the AORA process model. The approach, together with the composition language, was applied to a real case study in the air traffic control domain, the "Flight PLan (FPL) Tower Interface" system, which is operated by NAV[1], the company that controls the Portuguese air space.

The contributions of this paper are three fold: (i) providing an expressive composition language, (ii) defining a process to guide the composition activities, (iii) integrating the composition language grammar syntax into the existing AORA tool, refining the existing AORA Composition Rule Editor (CRE).

The remaining of this paper is organised as follows. Section 2 briefly presents the AORA context, motivation and approach. Section 3 presents the grammar syntax and approach for the Multi-Dimensional Composition by Objectives (Multi-ComBO, here simplified many times to *ComBO*) language composition. Section 4 presents the application of the proposed approach and language in an operational system. Section 5 presents the ComBO integration into the AORA tool. Section 6 discusses some related work. Finally, section 7 provides an overview evaluation, draws some conclusions and presents future work.

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, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE'08, May 10–18, 2008, Leipzig, Germany.

Copyright 2008 ACM 978-1-60558-079-1/08/05...$5.00.

[1] http://www.nav.pt/

2. BACKGROUND: ASPECT-ORIENTED REQUIREMENTS ANALYSIS

AORA is composed of three main tasks: (i) identify concerns, (ii) specify concerns and (iii) compose concerns. These tasks can be accomplished iteratively and incrementally, allowing each task to be performed without the full completion of the previous task. The AORA tool supports concerns and responsibilities specifications, identification of Match Points and crosscutting concerns, and supports the definition of composition rules [3, 4]. A **Responsibility** is considered as an obligation to perform a task, or some functionality that the system shall provide. A **Match Point** is composed of a set of concerns that need to be composed together. One of the concerns plays the role of base concern on which the behaviour of the remaining concerns needs to be weaved into. These concerns are the required concerns of the base concern[2].

The concepts are stored in XML using eXist [12] an open source, native XML database.

Due to space limitations, we invite the interested reader to refer to [1-4] for broader details of the AORA approach and tool.

3. COMPOSITION BY OBJECTIVES

3.1 Context

AORA's composition process follows a black-box concern-based approach, therefore offering coarse-grained composition rules. Also, it offers a reduced number of operators, consequently supporting low expressivity of the composed artefacts, which is needed when the software development moves to later stages. The approach proposed in this paper, the Multi-Dimensional Composition by Objectives (Multi-ComBO), refines the AORA composition task, allowing composition at a finer granularity level, whose driver is the objective or the purpose of the composition. Conceptually, the ComBO rules were defined to compose rules that allow the identification of potential concern interactions (e.g., conflicts) used to derive composed models that give an integrated view of the system, and define the order wherein concerns will be applied in a particular Match Point. The ComBO composition rules work at the concerns' responsibilities level.

Each composition reflects a purpose, or objective, and allows a Multi-Dimensional composition. By Multi-Dimensional we mean that each concern is a dimension and no concern classification is needed at this stage. Therefore concerns are treated similarly, promoting simplification of the initial representations and avoiding the tyranny of dominant decomposition [13]. The ComBO approach allows us to compose concerns with a diverse palette of operators.

Rules can be complex, and as composition rules can be built of others, by objective or goal, their scalability and complexity is also handled by ComBO. Consequently, this also means that new dimensions can arise from the original compositions' dimensions. As a result, Multi-Dimensional scalability is achieved through composition, as the initial dimensions do not dictate the final dimensions. Sets of rules can therefore be composed to achieve a specified objective. This rule composition approach is a considerable improvement since each rule has a real purpose and objective, instead of having composition of rules that are only composing artefacts together without giving explicitly any particular reason or utility for it. Besides, a minimum traceability is supported by the approach regarding the rule composition.

3.2 Language Syntax & Approach

A ComBO rule consists of a header and a body. The header introduces the concrete objective to be attained. It states the context where the body is to be applied. The body of the rule gives the formal composition rule, detailing the respective header rule. (The complete ComBO grammar syntax is presented in annex.)

3.2.1 *The ComBO header*

The header starts the composition rule and is followed by the body, as depicted in Figure 1. The rule header starts with the special expression `to achieve` (see Figure 2). The motivation for choosing this expression results from its semantics and the definition of the "achieve" verb: "*To perform or carry out with success, to accomplish*" [14] and its synonyms: "realize", "attain", and "reach" [15]. The `to achieve` expression is followed by a list of one or more objectives, that will be *achieved* following the rules described in the ComBO body.

Figure 1. The ComBO rule structure

The example in Figure 2 specifies that both the responsibilities **`ATM.activation`** *and* **`ATM.reactivation`** (Green Lane example, [7]) can be achieved by the same rule. What the rule expresses is: "***to achieve*** an objective ***or*** another, the rule in the ComBO body must be fulfilled".

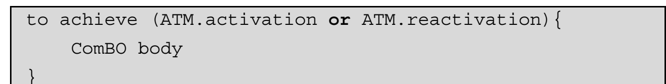

Figure 2. Instance for the ComBO rule header

The `term` expression represents functional and non-functional artefacts such as concerns and responsibilities. It is used in the ComBO header, in the list of objectives, and in the ComBO body. The ComBO **`term`** representation and respective grammar syntax is depicted in Figure 3.

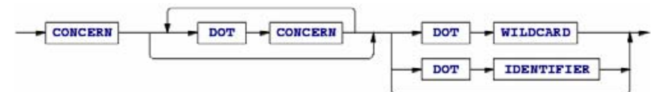

Figure 3: The `term` syntax graphical representation

To be able to compose rules, the `term` grammar syntax supports simple name representation. Its grammar syntax allows complex composition that some artefacts can require until achieving a wanted level of operationalization or abstraction, depending on the performed analysis. Similarly as the fine-grained analysis performed by Marques [16], which from resulted heavily compositions and decompositions of hierarchized concerns such as: *"Efficiency. Time_Performance. Response_Time. Real-*

---

[2] For instance, if concern C_1 requires C_2 and C_3, then the match point MP_C_1 is composed of C_1, the base, and C_2 and C_3, that is $MP_C_1 = \{C_1, C_2, C_3\}$.

Time_Processing_Response_Time" – from lower to higher granularity (and therefore abstraction) separated by dots ('.' character), this represents the non-functional concern (i) "*Real-Time Processing Response Time*", a decomposition of the non-functional concern (ii) "*Response Time*" itself a decomposition of the (iii) "*Time Performance"* concern which is itself a finer decomposition of the (iv) non-functional concern "Efficiency". The asterisk character ('*') is also available and can be used to express "*the addition or use of all the existing artefacts hierarchically structured below (and part of / belonging to) the current one*". The ComBO grammar syntactically supports identifiers naming (i) by numerical *id,* such as "R002", "resp003", and (ii) by *name*, being able to support several notations and improving the language expressiveness.

3.2.2 The ComBO body

Rules in the rule body only make sense to attain the (*list of*) specified objectives. As depicted in Figure 4, ComBO body rules can be (i) empty, in cases where to achieve some operationalizable artefact we *may* not need to compose any artefacts together; or (ii) composed by complex expressions (symbolized by the do_expression label) and statements (represented by the alt_statement label).

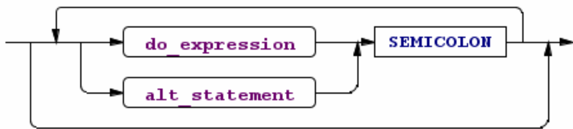

Figure 4. The ComBO body syntax structure

3.2.3 do_expression

The do_expression is used to express and confront the behaviours of artefacts between other artefacts (do_sequence structure). Although not mandatory, the do_expression additionally allows expressing temporal conditions (temporal_statement structure). Once the do_statement structure is grammatically recognized with the do keyword, follows a term or a break_expression (*breaking* expression). The breaking expression is equivalent to a halting instruction and is represented by the break keyword, and when employed, consequently resumes the rules at the next statement. Optionally, after the do_statement rule structure introduction, can follow a do_sequence rule structure. Figure 5 illustrates an example that makes use of the **do_statement** structure with hierarchies of concern composition: **gizmo.identification** and **ResponseTime.read_gizmo_identifier**. This means that "*to identify the user's gizmo, the reading of the gizmo identifier must have been fulfilled in a given fraction of time*".

```
to achieve (gizmo.identification){
    do ResponseTime.read_gizmo_identifier;
}
```

Figure 5. Example of a do_statement employment

The do_sequence grammar syntax structure is not mandatory and in the context of the do_expression structure takes place after the do_statement declaration. The do_sequence structure is used to express constraints among term expressions. These constraints can be after, before, instead of and between. The after and before constraint operators are related: the do-after constraint expresses the realization of artefacts *after* the execution of necessary artefacts. Analogously, the do-before constraint expresses the realization of artefacts *before* the remaining artefacts. The instead of constraint operator will substitute the artefact right after the instead of keyword, substituting a behaviour by another. The between keyword, Figure 6, is used to express temporal interval *between* two artefacts.

```
to achieve (gizmo.identification){
    do ResponseTime.read gizmo identifier between
vehicle.system_entrance and vehicle.toll_gate_entrance;
}
```

Figure 6. The between constraint operator example

In Figure 6 the temporal interval starts when the vehicle enters in the green lane system (**vehicle.system_entrance**) and ends when the vehicle is entering in the toll gate, i.e., ends when **vehicle.toll_gate_entrance** is being fulfilled.

Like the do_sequence structure, the temporal_statement structure is not mandatory. Its purpose is to make available temporal restriction expressivity upon the composition rule. Several constraints are available: starts, concurrent, finishes and through constraints – based on [7, 17] and [13]. The starts constraint is useful to express situations like the one in Figure 7, "*concern A **commences** simultaneously with concern B, then, concern A can completes its behaviour independently if concern B completes before or not*".

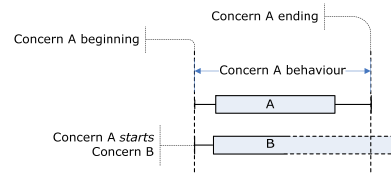

Figure 7. The *starts* constraint graphical representation

Figure 8 introduces an example: "*In order **to achieve** the gizmo's identification, the vehicle's toll entering will **concurrently trigger and start** the gizmo's authenticity verification*". It is necessary to consider that the concept of concurrency is *theoretically* introduced, but in reality there is always a minimum delay[3] to execute two or more processes simultaneously. This delay, as minimum as possible, denies the *practical* concurrency. Here, any practical restrictions are ignored, since the focus is on analysis, leaving the concurrency problems for the later stages.

```
to achieve(gizmo.identification){
 do vehicle.toll_gate_entrance
  during( starts( gizmo.identification.verification));
}
```

Figure 8. The *during* condition with the *starts* constraint

The concurrent constraint means that "*behaviour of concern A and C are **started and completed within the exact same temporal interval***", as depicted in Figure 9.

[3] And software and hardware restrictions, but those are not the scope of this work.

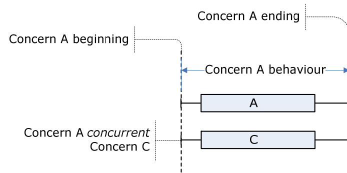

Figure 9. The *concurrent* constraint graphical representation

Figure 10 presents the example: "*in order **to achieve** the course through the toll, the toll must display notifications – accordingly with each (in) valid situation, **concurrently** as the vehicle exits the toll*".

```
to achieve(toll.through_passage){
    do toll.display_notification during (concurrent
(vehicle.toll_gate_exiting) );
}
```

Figure 10. A *during* condition with the *concurrent* constraint

The `finishes` constraint represents situations as the one depicted in Figure 11: "*concern D has commenced **before** the behaviour of concern A or **commences** while concern A is in progress, however **both complete simultaneously***".

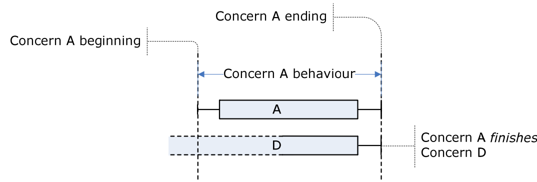

Figure 11. The *finishes* constraint graphical representation

Figure 12 depicts an additional example: "*in order **to achieve** the verification of the gizmo's identification, the gizmo's verification must be done **as** the gizmo's identification is done, **finishing** this last action, once it is concluded*".

```
to achieve(gizmo.identification.verification){
    do gizmo.verification during( finishes(
gizmo.identification));
}
```

Figure 12. The *during* condition with the *finishes* constraint

Finally, the `through` constraint is useful to express several situations – as depicted in Figure 13, like "*concern E has **commenced before** concern A, concern A commences while the behaviour of concern E is **still enabled**, or the behaviour of concern A **completes** while the behaviour of concern E is **still enabled***".

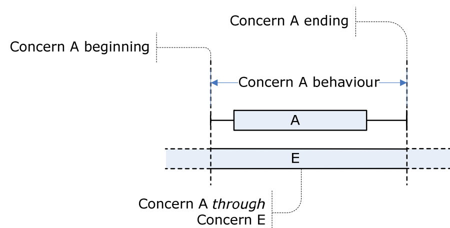

Figure 13. The *through* constraint graphical representation

Figure 14 illustrates the `through` constraint: "*in order **to achieve** a proper notification from the toll, the toll must display any notification **as** the vehicle is exiting the toll gate.*"

```
to achieve(toll.notification){
    do toll.display_notification during ( through(
vehicle.toll_gate_exiting));
}
```

Figure 14. The *during* condition with the *through* constraint

Figure 15 introduces a temporal statement associated to a list of restrictions. The **`temporal_statement`** can support constraints to express concurrency between artefacts. Figure 15 illustrates this (Pre-Start Engine Heat System, [18]): "***to achieve** the engine heating of a vehicle, one **must** turn the vehicle key. The act of turning the key will **start** the heating of the engine's air **and** furthermore the vaporizing of the engine's fuel. Once the key turning is done, the engine's electrical heating **finishes***".

```
to achieve (engine.heat){
    do vehicle.key turning during (starts
(engine.fuel vaporizing, engine.air heating), finishes
(engine.electrical_heating));
}
```

Figure 15. Example of a `temporal_statement` restriction

Several variation of the `temporal_statement` grammar syntax are available: (i) by time interval, (ii) to express concurrency between artefacts, (iii) and both, by confronting the execution time of an artefact, useful to the composition of rules that need to express concurrency between artefacts with delimited amount of time.

3.2.4 alt_statement

The language rule for the ComBO syntax's `do_expression` allows condition statement through *alternatives*. Those can be expressed by the `alt_statement` ComBO body syntax rules. The `alt_statement` conditionally executes a (*list of*) `do_expression` statement, depending on the value of a (*list of*) Boolean expression.

Figure 16 illustrates the use of the `alt_statement` structure. The introduced condition is relative to the case that a given member possesses a "Gold" card: if true, a "gold member" welcome treatment will be applied; else a "regular member" welcome will take place. In the example, the list of Boolean conditions has only one condition, `member.cardType == "Gold"`, but could have more than one. Additionally to Boolean testing with strings between artefacts, it is also possible to do so with *numerical compositions*.

```
to achieve (member.welcomeMember){
    alt(
        cond member.cardType == "Gold";
        do member.welcomeGoldMember;
        else do member.welcomeRegularMember
    );
}
```

Figure 16. Use of the `alt_statement` conditional rule

A `numerical_composition` can be either simple (e.g., `1 + 2`) or complex in expressivity (e.g., `3 * (timer + counter)`). A `numerical_composition` is ruled by a `numerical_expression`, which is either a literal numeric value or a term – assumed to be the identifier of a numeric value. Logical symbols and (in) equality operators ('`==`', '`!=`') are allowed.

3.3 The ComBO Approach

The ComBO approach is incremental and iterative, following two simple steps: *Local Rule composition* and *Final Rule composition*. Figure 17 depicts the composition rule process overview. The Local Rules are made for each concern composing the Match Point in analysis. For example, if a given Match Point named "MP_FlightStripPrinting" with base concern Flight Strip Printing, requiring the Retrieval, Logging, Correctness and Update concerns, there shall be created one rule for each of the listed concerns.

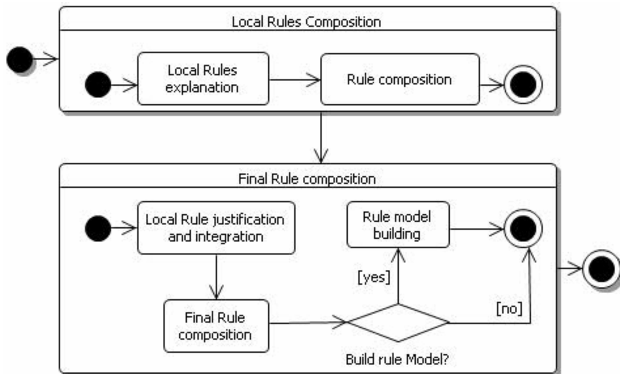

Figure 17. Global composition rule process overview

Once the Local Rules are composed, they are then merged together. The last main activity is to create the Final Rule composition, resulting from the combination of the previously defined Local Rules.

3.3.1 Local Rules

Local Rules are intended to help on the creation of the Match Point's final composition rule, since Match Points can have a high number of concerns[4]. In this way the *Local Rules* anticipate the final composition by initially involving the software engineer in the concerns that composes the Match Points, leading to an (i) easier composition rule creation [19], (ii) increased understanding, and (iii) enabling traceability of the created rules. The number of *Local Rules* is equal to the number of the analysed concern's responsibilities that are interacting with the Match Point's base concern's responsibilities. If a given concern *C* has responsibilities *RespX* and *RespY,* both with *responsibilities* towards the Match Point in analysis, composing Match Point *M* with base concern *B*, then, for the *C* concern's analysis, there should be created two Local Rules.

3.3.2 Final Rules

The *final composition* first sub-step takes form from the *local justification and integration* iterations. This step is necessary to justify how and why the Local Rules have been composed (to help on traceability). The Final Rule may be decomposed into more than one rule, representing different behaviours.

This process is exemplified by means of a case study from the SOFTAS project in Section 4.

[4] In small and large systems.

4. CASE STUDY

The Flight Data Processing System (FDPS) is a case study of the SOFTAS project (POSI/EIA/60189/2004) proposed by NAV, a partner of the project. FDPS is responsible for the flight data processing in Portugal's aerodromes. Most of the data used by the system is obtained from the Airport Operational System (AOS), Flight Data Section (FDS) and Environment. The system offers four main groups of functionalities: Airdrome, Departure, Arrival and Overflight, all connected through a central application. In each of these window groups, it is possible to list, visualize and modify (if necessary) the respective item parameters. In case of departure and arrival flights, it is also possible to print paper strips. Figure 18 presents the Local Rule for the Retrieval concern of the Flight Strip Match Point: this composition means that the body of the rule will enable the accomplishment of **all** the responsibilities of the Flight Strip Printing concern, namely: **`PrintArrivalData`**, **`PrintDepartureData`** and **`PrintOverflightData`**. The rule can be justified as: "*To be able to print data, the data must have been already retrieved*".

```
to achieve(FlightStripPrinting.*){
    do Retrieval.RetrieveFSPData;
}
```

Figure 18. The Retrieval Local Rule

Figure 19 presents the ComBO Local Rules for the Correctness concern of the Flight Strip Printing concern. The composition presented is for the Print Arrival Data responsibility, and can be justified as: "*Before printing any Flight Strip, the arrival data to be printed must be correct*".

```
to achieve(FlightStripPrinting.PrintArrivalData){
        do Correctness.EnsureFSPDataSpecification;
}
```

Figure 19. The Correctness Local Rule for *PrintArrivalData*

Figure 20 depicts **partially** the Final Rule for the Flight Strip Printing concern: the rule includes one of its responsibilities, the *PrintArrivalData*.

```
to achieve ( FlightStripPrinting.PrintArrivalData ){
    do Correctness.EnsureFSPDataSpecification during(
        through (Update.UpdateArrivalData),
        through (Retrieval.RetrieveFSPData)
    );
    do Logging.PrintFS during(
        through (Update.UpdateArrivalData),
        through (Retrieval.RetrieveFSPData)
    );
    do Retrieval.RetrieveFSPData;
    do Update.UpdateArrivalData;
}
```

Figure 20. Partial Final ComBO Rule for *FlightStripPrinting*

The justification given, for (i) traceability purposes and (ii) improvement of the understanding of the created rule, in the context of this case study and for the Flight Strip Printing, is that the Correctness concern must be assured for all activities, essentially during Flight Strip Printing. Therefore: (i) with correctness, all events are logged; (ii) logging of an event must be done during retrieval and update operations; (iii) retrieval of data can then be executed; (iv) before the Flight Strip Printing

operation, *PrintArrivalData* in this case, updates are required (`do Update.UpdateArrivalData;`).

This small extract of the case study shows that our composition approach helped in enriching the requirements document by specifying more detailed relationships between concerns which would normally appear at later stages, when it would be more difficult to handle due to the complexity of the late requirements or design models.

Due to space restrictions and legal issues we presented here a small part of the case study and its results.

5. TOOL SUPPORT

The Java parser generator of the ComBO language was created using the Eclipse platform version 3.2 [20] and the Eclipse plug-in version 1.5.7 [21] of Java Compiler Compiler (JavaCC) version 4.0 [22]. The integration of the parser into the AORA tool was implemented in the Netbeans IDE version 4.1 [23] on top of Java JDK 1.5 [24].

Although all the existing features of the AORA tool were preserved, it has suffered some modifications to be able to assess the ComBO rules and approach. In particular, it was necessary to develop: (i) HTML report conversion, (ii) basic and automatic graphical generation of the composed rules, (iii) logging support, for the history functionality, (iv) Composition Rule persistent storage in the eXist repository, such as reading and writing, (v) concern creation and update and major refinements in the Composition Rule Editor (CRE). The CRE interface added support for handling Local and Final Rules (insert/delete/view). Obviously, we needed to upgrade the grammar syntax parser and the AORA rule interpreter with the new ComBO grammar in order to verify the rules insertion. Figure 21 briefly synthesizes the composition rule process.

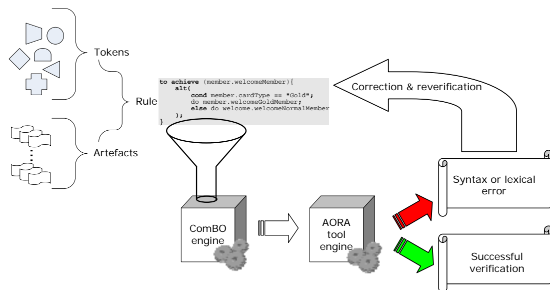

Figure 21. Composition rule phases

Tokens and artefacts *assembled* together construct a rule: the tokens are the syntactic connectors of the artefacts. Once a rule is composed it is then parsed by the ComBO engine which notifies the AORA tool interpreter of syntactic errors. The AORA interpreter also checks for lexical errors, such as unknown artefacts in the current context. After a rule is verified by the two engines, it can be (i) successfully verified, i.e., no syntactic or lexical errors where reported, or (ii) errors where found: the next step is to correct the composed rule and restart all the previously performed iterations.

6. RELATED WORK

This work mainly results from the detection of negative and election of positive details from other works of the most relevant approaches, which propose some kind of composition between concerns at the requirements analysis level.

The composition rules presented in [7] do not propose a rigorous process to compose concerns together but uses actions and operators to specify how an aspectual requirement influences the behaviour of a set of non-aspectual requirements. This process of modularisation enables the early identification of trade-offs between aspectual requirements and consequently affords the necessary maintainability for negotiation and to create a consensus among the involved stakeholders.

The Requirements Description Language (RDL) [25] is based on the natural language itself to support definition of a flexible composition mechanism for requirements analysis. RDL is based on the symmetric view of AOSD [8, 26-28] and uses the same abstraction as [7]: concerns, to represent both crosscutting and non-crosscutting elements, descriptions of requirements, intention of the system, the system behaviour, and constraints and standards proposed by Sommerville [29].

Study and addressing of characteristics in existing approaches allowed us to consider that from [7] the consideration of an *Outcome* for the composition of artefacts is useful as it states a post-condition of a composition. [1, 2] and [8] also based their approaches on the Multi-Dimensional Separation of Concerns. [1, 2] and [25] give importance to temporal composition, also known as aspect interaction [30]. Authors of [28] also motivate the need for multiple dimensions and rich mechanisms for composition at later stages of software development.

7. CONCLUSIONS

A requirements engineer will always have to reason about the contextual requirements of systems: for example, tools can help on verifying compositions, but it is the work of the software engineer to validate the composition. A composition environment, as the one proposed here, facilitates this process.

The AORA v1.0 had few operators and its compositions were defined at a too coarse-grained level of granularity. Some other problems, for example those detected in [25], have also been addressed, such as support for name identifier. Therefore, our approach, that extends AORA composition and its tool to v1.1, attempts to offer another contribute in AORE, by providing a novel expressive composition rule at the early-requirements stage. As future work, we will apply the approach to more case studies.

Acknowledgments: This work was partly supported by the Portuguese Fundação para a Ciência e Tecnologia, in the context of the SOFTAS project (POSI/EIA/60189/2004).

8. REFERENCES

1. Brito, I.S., et al., *Handling Conflicts in Aspectual Requirements Compositions*, in *Transactions on Aspect-Oriented Software Development, Special Issue on Early Aspects*. 2007.

2. Brito, I.S. and A. Moreira, *Towards an Integrated Approach for Aspectual Requirements*, in *Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06)*. 2006, IEEE Computer Society.

3. Soeiro, E., I.S. Brito, and A. Moreira, *An XML-Based Language For Specification And Composition Of Aspectual*

Concerns, in *8th International Conference on Enterprise Information Systems (ICEIS 2006)*. 2005: Paphos, Cyprus.

4. Brito, I.S., A. Moreira, and J. Araújo, *Tool Support for Aspect-Oriented Requirements*, in *The 10th IASTED International Conference on Software Engineering and Applications*. 2006, IASTED: United States.
5. Linda, A.M., *Requirements engineering*. 1996: Springer-Verlag.
6. Richard, H.T., C.B. Sidney, and M. Dorfman, *Software Requirements Engineerings, 2nd Edition2*. 1997: IEEE Computer Society Press.
7. Rashid, A., A. Moreira, and J. Araújo, *Modularisation and Composition of Aspectual Requirements*, in *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, AOSD 2003*. 2003, ACM Press: Boston, Massachusetts.
8. Moreira, A., J. Araújo, and A. Rashid, *A Concern-Oriented Requirements Engineering Model*, in *Proceedings International Conference on Advanced Information Systems Engineering (CAiSE) 2005*. 2005, LNCS. p. 293 - 308.
9. Jacobson, I., et al., *Object-Oriented Software Engineering – a Use Case Driven Approach*, in *Addison-Wesley*. 1992.
10. Baniassad, E. and S. Clarke, *Theme: An Approach for Aspect-Oriented Analysis and Design*, in *Proceedings of the 26th International Conference on Software Engineering*. *ICSE*. 2004, IEEE Computer Society.
11. Logrippo, L., M. Faci, and M. Haj-Hussein, *An introduction to LOTOS: learning by examples*. 1992, Elsevier Science Publishers B. V. p. 325-342.
12. Meier, W. *eXist XML Database Management System*. 2002. [cited; Available from: http://exist.sourceforge.net/.
13. Bakker, J., B. Tekinerdogan, and M. Aksit, *Characterization of Early Aspects Approaches*, *International Workshop on in Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, AOSD 2005*. 2005: Chicago, Illinois, USA.
14. *The American Heritage Dictionary of the English Language*, ed. E.o.T.A.H. Dictionaries. 2005: Houghton Mifflin; Thumb Indexed edition.
15. Thinkmap. *Thinkmap Visual Thesaurus*. 2007 [cited; Available from: http://www.visualthesaurus.com/.
16. Marques, A.G., et al., *Aspect-Oriented Analysis Applied to the Space Domain*, in *ICEIS 2007 – 9th International Conference on Enterprise Information Systems*. 2007: Funchal, Portugal.
17. James, F.A., *Maintaining knowledge about temporal intervals*. 1983, ACM Press. p. 832-843.
18. Sarto, J.O., *Pre-start engine heat system* 1983, CHRYSLER CORP: Orchard Lake, Michigan, U.S.A.
19. Miller, G.A., *The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information.* The Psychological Review, 1956. **63**: p. 81-97.
20. Eclipse Foundation. *Eclipse Download*. 2007 [cited; Available from: http://www.eclipse.org/downloads/.
21. Koutcherawy, R. *JavaCC Eclipse Plugin*. 2007. [cited; Available from: http://pagesperso-orange.fr/eclipse_javacc/.
22. *javaCC: JavaCC Project Home*. 2007. [cited; Available from: https://javacc.dev.java.net/.
23. *Netbeans home page*. 2007. [cited; Available from: http://www.netbeans.org/.
24. Sun Microsystems. *Sun Microsystems - Sun Developer Network (SDN) - Downloads*. 2007 [cited; Available from: http://developers.sun.com/downloads/.
25. Chitchyan, R., et al., *Semantics-based composition for aspect-oriented requirements engineering*, in *Proceedings of the 6th international Conference on Aspect-oriented software development, AOSD 2007*. 2007, ACM Press: Vancouver, British Columbia, Canada.
26. Moreira, A., R. Awais, and J. Araújo, *Multi-Dimensional Separation of Concerns in Requirements Engineering*, in *Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE'05) - Volume 00*. 2005, IEEE Computer Society: Paris, France.
27. Stanley M. Sutton, Jr. and I. Rouvellou, *Modeling of Software Concerns in Cosmos*, in *Proceedings of the 1st international Conference on Aspect-Oriented Software Development, AOSD 2002*. 2002, ACM Press: Enschede, The Netherlands.
28. Tarr, P.L., et al., *N Degrees of Separation: Multi-Dimensional Separation of Concerns*, in *ICSE*. 1999, ACM: Los Angeles, USA. p. 107-119.
29. Sommerville, I., *Software Engineering: (Update) (8th Edition) (International Computer Science)*. 2006: Addison-Wesley Longman Publishing Co., Inc.
30. Bakre, S. and T. Elrad, *Scenario based resolution of aspect interactions with aspect interaction charts*, in *Proceedings of the 10th International workshop on Aspect-Oriented Modelling*, AOSD 2007. 2007, ACM: Vancouver, Canada.

9. ANNEX

```
to achieve (<list_of_objectives>){
    do artefact
     ([ (<OP 1> artefact b) | (between artefact c
(<OP_2> artefact_i) *) ])?
     ( during [ time interval | (<OP 3>
list_of_artefacts (,<OP_3> list_of_artefacts )* ) |
artefact_b <OP_4> time_interval ] )?;
    alt(
        cond <list_of_boolean_expression>;
        <list_of_condition_rule>;
        else <list_of_condition_rule>);
}
where:
<list of objectives> := <term> (or <term> | and not
<term_i>, <term_ii>, …, <term_n> )*
<term:> := [concern ('.'concern)+ ['.*'|'.'identifier]]
<list of boolean expression> := <boolean expression> (,
<boolean_expression>)*
<boolean expression> :=[(<numerical composition><OP 5>
<numerical_composition>) | (artefact <OP_6> string) ]
<numerical composition> := [ <numerical expression> |
(<numerical_expression><OP_7><numerical_composition> )]
<numerical_expression> := [ artefact | integer ]
<list_of_condition_rule> := <condition_rule>
(,<condition_rule>)*
<condition_rule> :=[<break_expression>|<do_expression>]
<break_expression> := break
<OP_1> := [after | before | instead of]
<OP_2> := [and | or]
<OP_3> := [starts | concurrent | finishes | through]
<OP_4> := [< | <= | = | > | >=]
<OP_5> := [< | <= | == | != | > | >=]
<OP_6> := [== | !=]
<OP_7> := [+ | - | * | / | %]
```

Figure 22. The complete ComBO grammar syntax