Managing concern interactions in middleware

Frans Sanen, Eddy Truyen, and Wouter Joosen

DistriNet, Department of Computer Science, K.U.Leuven,
Celestijnenlaan 200A, 3001 Leuven, Belgium
{Frans.Sanen, Eddy.Truyen, Wouter.Joosen}@cs.kuleuven.be

Abstract. In this paper, we define a conceptual model that describes the relevant information about interactions between concerns that needs to be captured. We have developed a prototype system that, starting from this model, can automatically generate a set of rules that enables software developers to improve their understanding of concerns in middleware and their interactions. This rule-base is the basis for an expert system that can be queried about particular concern interactions and a software engineering tool to support an application development team.

1 Introduction

In this paper, we present a conceptual model that helps a software development team to understand and manage the different interactions between typical concerns in a component-based distributed application. In general, the conceptual model complements methods for building large and complex component-based distributed systems. To demonstrate results, we have focussed on the particular application domain of middleware (see Section 2).

Nowadays, software applications are being increasingly complex and large-scale. This is mainly because of two reasons. First, the number of different implemented concerns has exploded. Concerns are similar to requirements in a broad sense of the word (a more detailed definition of concerns will be given in Section 2). Secondly, and more importantly, there are many (often hidden) interactions between all these different concerns. Concerns are typically not completely orthogonal to each other, but can relate to each other in a variety of different ways: they can either depend upon each other, conflict with each other, exclude each other, etc... This makes it challenging for a development team to understand, sustain, maintain, adapt and evolve contemporary software applications.

There is a wide consensus in the software engineering community that in order to manage large and complex software systems, one must rely on intensive separation of concerns [29] and componentization [1]. Separation of concerns is not easy to achieve however. Software development methods often only achieve a good separation of concerns if sufficient application domain expertise is present within the development team. We therefore believe that application domain expertise should be represented explicitly as part of a conceptual model such that it can be shared and used during the course of system evolution. The conceptual model we propose identifies and describes the different concerns of importance in the middleware domain and provides the foundation for the construction of reusable components.

This paper addresses the following problems. Expertise about interactions between concerns (and therefore components) is seldom made explicit. As a result, this knowledge cannot be shared and used among a development team. In monolithic software, concern interactions are often hidden in the implementation details of components. Yet, we argue that concern interactions and expertise on how to resolve these also forms very important domain knowledge as this leads to a better understanding of the application domain. Despite the fact that modeling techniques exist that explicitly represent concern interactions (e.g. [9, 11]), few of these modeling techniques make the link between concerns and implementation components, i.e., they provide no practical support to the software developer for

managing the interactions when he/she is in the course of creating, adapting or evolving the component composition of an application. This problem statement will be elaborated upon in Section 2 using some scenarios.

Our contributions are threefold. Next to the conceptual model for representing concern interactions we also propose to use reasoning techniques for detecting interactions in a given concern composition. Third, as a proof of concept, but also as a useful tool, we have implemented a solution in OWL [28] and Prolog [31]. The CIA (Concern Interaction Acquisition) expert system uses Prolog as a reasoning technique for detecting interactions in a given concern composition. The acquisition of the interaction knowledge to be captured in the conceptual model is realized through OWL. We consider this expert system as a backend for various software development tools.

The rest of this paper is structured as follows. Section 2 motivates our research by setting the scene and elaborates on some specific concern interactions in middleware we want to investigate. We explain the proposed conceptual model in Section 3 and illustrate it with a running example. Section 4 discusses the prototype of the CIA (Concern Interaction Acquisition) expert system. Finally, related work is discussed in Section 5. We present a conclusion in Section 6.

2 Background and motivation

2.1 Background

Management of concern interactions is a general problem that is relevant in many application domains, such as telecommunications, middleware, email, thermo control, policy-based, multimedia and other systems [5–7, 12, 17, 20, 23, 24]. Our work is focused on the domain of *common middleware services* and all example concern interactions and further details within this paper should be interpreted with respect to that background.

Middleware is systems software that resides between the applications and the underlying operating systems [38]. Its primary role is to functionally bridge the gap between application programs and the lower-level and heterogeneous software infrastructure. It is used most often to support complex, distributed applications. It includes web servers, application servers, content management systems, and similar tools that support application development and delivery. Middleware is typically decomposed into four layers [38], which are shown in Figure 1.

- Host infrastructure middleware provides an abstraction layer that shields software in the higher layers from the details of the underlying OS (Operating system). By abstracting away the peculiarities of individual operating systems, many tedious and error-prone aspects of sustaining networked applications via low-level OS programming APIs are eliminated. Widely known examples are Sun's Java Virtual Machine [22], Microsoft's .NET [40] and the ADAPTIVE Communication Environment [39].
- Distribution middleware defines higher-level distributed programming models whose reusable APIs and components automate and extend network programming capabilities encapsulated by host infrastructure middleware [38]. One advantage that is most cited for this middleware layer is that it provides networking transparency to the programmer. CORBA ORB's [26] and RMI [43] are two well-known examples.
- Common middleware services are built upon distribution middleware. They define higher-level domain-independent services that allow application developers to concentrate on programming business logic. Without these services, end-to-end capabilities (such as transactional behavior, security, database connection pooling or threading) would have to be implemented ad hoc by each networked application over and over again. The form and content of these services will continue to evolve as the requirements on the applications being constructed expand. Logical examples here are CORBAservices [27], EJB technology [15] and .NET web services [40].

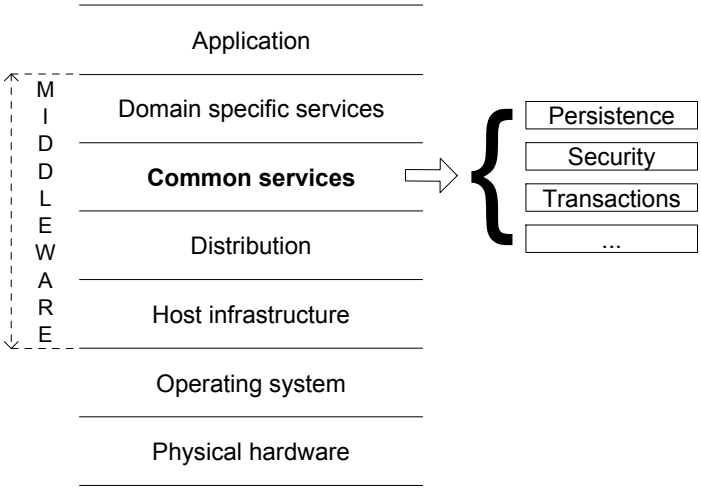

Fig. 1. The different layers in common middleware.

– Domain-specific middleware services are tailored to the requirements of particular domains, such as telecom, e-commerce, health care, process automation, or aerospace. Unlike the other three middleware layers, which provide broadly reusable horizontal mechanisms and services, domain-specific middleware services are targeted at vertical markets.

2.2 Motivation

What are concerns in middleware? Concerns are similar to requirements in a broad sense of the word, ranging from high-level requirements that are articulated in an early stage of the software project[1] to additional - often more detailed - requirements that are generated when performing detailed design and implementation[2]. Moreover, the various concerns embodied in current middleware can be situated from the lower-level host-infrastructure and distribution software layers to the higher-level common and domain-specific middleware services. For the sake of understanding, we use the term concern throughout this paper both for requirements and the artefacts that realize or implement these requirements in later stages of the software development lifecycle.

Common middleware services such as security, persistence and others correspond naturally to a number of (mostly non-functional) concerns that typically can interact with each other. Notice that there exist different sorts of interactions. We elaborate on a classification of concern interactions in Section 3. We now discuss three motivating examples of concern interactions. We provide some more detail regarding the common middleware services that are involved as it is not our intention to come up with a complete and exhaustive overview of existing common middleware services.

– In most cases, it is useless to have an authorization service without an authentication service. Authentication is the confirmation of a claimed set of attributes or facts with a certain level of confidence by providing sufficient evidence thereof. For example, providing your user name and password to your email client is a possible way to authenticate a person, principal or entity. Authorization refers to (1) the permission of an authenticated entity to perform a defined action or to use a defined service or resource; (2) the process of determining, by evaluation of applicable permissions, whether an authenticated entity is allowed to have access to a particular resource.

[1] E.g. the middleware should ensure confidentiality when information is exchanged between two parties.
[2] E.g. decrypted messages should never be cached.

Authorizing an entity E to perform an action A only makes sense if you are sure that entity E effectively is entity E, i.e. entity E is authenticated. In other words, authorization depends on authentication.

- Audit and confidentiality services are in conflict with each other. An audit service is responsible for maintaining an audit trail, i.e. a record of events, in order to be able to trace the activities and usage of a software system. Confidentiality refers to the state of keeping the content of information secret from all entities but those authorised to have access to it. An often used mechanism to realize confidentiality is encryption, the process of obscuring information to make it unreadable without special knowledge. Suppose data item X has to be kept secret, i.e. is confidential. Is the goal to have only the encrypted version to be logged in case of an event in which the data item X is involved and hence, compromising the readability and usefulness of the audit track? Or should the audit trail just refer to the plain data item X, sacrificing a large part of its confidentiality. Clearly, both middleware services correspond to conflicting concerns. The same conflict also arises between caching and confidentiality. Caching refers to saving recently accessed data in a small fast memory in order to speed up subsequent access to the same data.
- A transaction and authorization service also possibly conflict. A transaction service handles units of interaction in a coherent and reliable way. Such units of interaction have to happen in an all-or-nothing mode and must be either entirely completed or aborted. An ideal transaction service guarantees all of the ACID (Atomicity, Consistency, Isolation and Durability) properties for each transaction. Suppose an entity starts a transaction. Consider for example the case where an employee starts to update some parts of the personnel database of the company he is working for. As soon as he tries to upgrade his monthly salary, the authorization service halts the execution by indicating the denied permission. The transaction service hence should no longer complete the started unit of interaction and abort.

We propose to explicitly capture this kind of interaction knowledge using a conceptual model that captures the most important concepts for representing concern interactions (see Section 3). The CIA expert system we propose in Section 4 will enable exploiting this knowledge as depicted in Figure 2. First of all, different domain experts incrementally insert new interaction knowledge coming from their domain expertise into the CIA system, based on the conceptual model. This corresponds to arrow (1) in Figure 2.

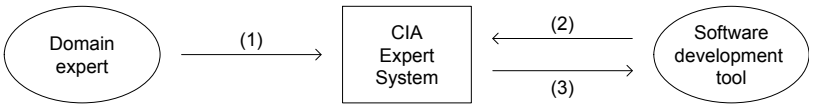

Fig. 2. Exploiting concern interaction knowledge.

Then, the expert system can be used in various composition activities during the software development process, such as for example trade-off analysis [35], component deployment [8] and concern composition. In this paper, we further investigate component composition during application assembly by means of a visual software composition tool. Suppose the user is creating a component composition using a visual composition environment. As the user drags new components on the canvas, the composition tool may query the CIA system about known interactions (see (2) in Figure 2). CIA then uses reasoning techniques to detect the interactions that occur in the given component composition. It will respond to the software composition tool with this list of interactions and tactics for resolving them (3). The composition tool may present then this knowledge to the user in its own notation. In the long run, a component framework that can cope with adding components in a flexible and generic way

could interpret these tactics automatically. The internals of the CIA expert system will be discussed in Section 4.

3 Conceptual model

In this section, we present the conceptual model behind the CIA system. We start with a general overview of the most important and top-level concepts in Section 3.1. In the subsections thereafter, we will provide some more detail on some of these concepts. Throughout the elaboration of our model, we will use our second motivating example from section 2, the conflict between audit and confidentiality services, to illustrate the different concepts and relationships between these concepts.

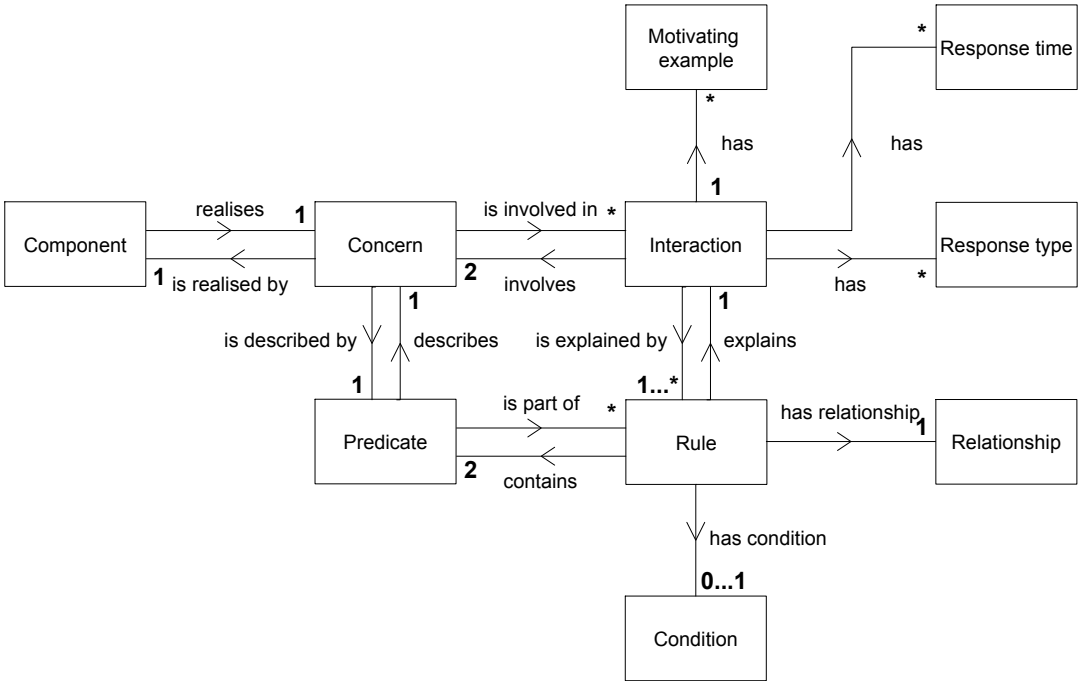

Fig. 3. A conceptual model for describing concern interactions.

3.1 Overview

Our conceptual model provides a number of concepts in terms of which knowledge about interactions between concerns can be acquired; it is thus a meta-model. It is aimed at being sufficiently rich to allow all kinds of concern interactions for any kind of component composition to be captured in a precise and natural way. The model for capturing concern interactions knowledge can be represented as a conceptual graph where nodes represent concepts and edges represent structuring links, similar to [11]. Figure 3 illustrates the most important portion of the conceptual model. Roughly spoken, a concern, which is realised through one component, can be involved in one or more interactions. Such an interaction is explained by one or more rules, which essentially are a relationship (indicating the kind of interaction) between a number of predicates that in turn each describe one concern.

Concerns The central concept in our conceptual model is *concern*. As been said before, concerns are similar to requirements in a broad sense of the word that are realized through one or more components, ideally one. Our model currently only deals with applications that implement a one to one mapping between components of concerns. In the model, concerns are organised based on subconcern refinement relationships into a specialization hierarchy. Hence, under the umbrella of this abstract concept are the more concrete concerns and their subconcerns, in our case the common middleware services and their subservices. Multiple inheritance within this concern specialization hierarchy is inherently supported because of the underlying mechanism we use to implement the conceptual model (see Section 4.2). We will discuss here only a small part of this concern hierarchy in order to illustrate the approach. What is important is that this concern hierarchy has to be based on a lot of domain expertise from within the different domains to reach an as complete as possible state. At the second level (the concern concept forms the root of the concern hierarchy), we distinguish between the different layers in middleware: host infrastructure middleware, distribution middleware, common middleware services and domain-specific middleware layers. In a third level, each of these is further refined. E.g. the common middleware services node is refined to specific common middleware services such as security, persistence, transactions, etc. Additionally, each of these can be again decomposed. As an example, standard security typically breakdowns into an authentication, authorization, audit, confidentiality, integrity and non repudiation service [37]. The discussed part of the concern hierarchy is shown in Figure 4.

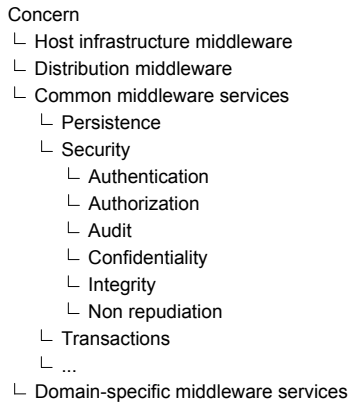

Fig. 4. Portion of the concern hierarchy.

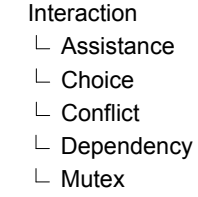

Fig. 5. Classification of concern interactions.

Interactions A concern can be involved in an arbitrary number of *interactions* with one or more other concerns. To structure concern interactions and address their effective management, we have devised a classification that distinguishes between different kinds of interactions. This interaction classification is based on earlier work and intensive workshop discussions [36, 3] and is shown in Figure 5. We distinguish between five different classes.

- Dependency covers the situation where one concern explicitly needs another concern and hence depends on it. A dependency does not result in a problem or erroneous situation as long as the concern on which another one depends is ensured to be present in the final component composition. E.g. authorization depends on authentication.
- Conflict captures the situation of semantical interference: one concern that works correct in isolation does not work correctly anymore when it is combined with other concerns. In other words,

a concern influences the correct working of another one negatively. Typically, a conflict can be solved by mediation or performing a trade-off analysis because the concerns, in a sense, are complementary. E.g. confidentiality and audit are conflicting (cfr. Section 2.2).
- Choice defines the interaction between two equivalent concerns. In other words, there is no need to have the components realizing both concerns deployed because their net effect will be the same. However, doing so won't give any problems. E.g. one of multiple authentication services gets chosen.
- Mutex encapsulates the interaction of mutual exclusiveness of concerns. Realizing one of both concerns prohibits the use of the other one. No mediation is possible because the concerns are not complementary: only one of them can be used, the other cannot. E.g. an extensive audit can compromise a certain strong timing constraint.
- Assistance arises when a concern influences the correct working of another concern positively and hence assists it. There can be no doubt that this type of interaction is a positive one. Typically, when a concern assists another concern, extended functionalities become possible and extra support is offered. E.g. caching encrypted data assists confidentiality by improving performance.

Predicates The third essential relationship for a concern shown in our conceptual model is the fact that a concern will be described by a *predicate*. Predicates enable us to describe the semantics of a specific concern. A predicate always has the format illustrated below. The definition of each predicate consists of two parts: a head, indicating the concern that is being described, and a number of parameters that are used to add all the concepts and values that are relevant for a complete description of a specific concern. Example definitions of the audit and confidentiality concerns in pseudo-code are shown in lines (2) and (3). (2) describes the audit concern in terms of four information items or parameters: entity, action, object and result. At all times, the semantics of the predicate is that all events where an entity `Entity` (e.g. a user) performing an action `Action` (e.g. a read operation) on an object `Object` (e.g. a data item) resulted in `Result` is recorded. Similarly, the confidentiality predicate can used to express that an object `Object` (e.g. a data item) is confidential.

```
(1) <head> ( <param1>, ..., <paramN> )
(2) audit ( Entity, Action, Object, Result )
(3) confidentiality ( Object )
```

An important characteristic of these predicates is their language and technology independence. Moreover, if we start from the observation that for each predicate, including for its head and parameters, an unambiguous definition exists, our approach is very intuitive. On the contrary, a mapping from this conceptual level to a lower language specific level will be needed in future work to ensure that a predicate at all times reflects the correct runtime state of the application components and the middleware environment.

Rules As depicted in Figure 3, our idea is to have each interaction between concerns explained by one or more *rules*. A rule essentially is a *relationship* between two predicates. It explains the context in which a specific interaction occurs by means of these predicates. Secondly, it also indicates the kind of interaction depending on the relationship a rule is associated with. There are five relationships included in our conceptual model corresponding to the five kinds of interactions we defined:

- depends_on, indicating a dependency,
- conflicts_with, indicating a conflict,
- mutex, indicating a mutual exclusion,
- one_of, indicating a choice, and
- assists, indicating an assistance.

We also take into account the concept of an optional *condition* which enables us to take certain conditions into account when describing an interaction. For example, if a conflict only appears under the runtime circumstances where battery power is low (as in [36]), we can express that. If we now look at our example, the interaction between audit and confidentiality clearly is a conflict, because both services operate correctly in isolation, but when they are composed, mediation is necessary to regulate their coexistence. Therefore, the rule explaining the interaction in pseudo-code is

```
for each Object o: audit(_,_,o,_) conflicts_with confidentiality(o)
```

It states that when certain objects (such as data items) both are required to be confidential and be part of events that need to be audited there is a conflict between audit and confidentiality.

Others Next to the information elaborated upon above, we obviously are also interested in possible solutions of specific interactions. This information need is reflected into the concepts *time of response* and *type of response*. It is clear that some stages of the software development lifecycle are more appropriate than others when trying to cope with a certain interaction. E.g. dependencies typically will be taken care of during the architecture phase, while conflicts potentially occur at runtime and need to be handled in later stages if they were overlooked during requirements analysis. The type of response concept represents the information definining an appropriate response to a specific interaction into more detail. Both can be considered as a sort of tactics for solving specific interactions. The details for specifying such tactics are subject of ongoing work. For now, a tactic compares to a textual description of the different alternatives to resolve the interaction. The remaining *example* concept is used to illustrate an interaction with a concrete motivating example for a better understanding of a specific interaction.

4 CIA expert system

In this section, we sketch the main architectural building blocks of the CIA (Concern Interaction Acquisition) expert system, again illustrated with our running example of the conflicting audit and confidentiality services. In Section 4.1, we present the high-level architecture of our expert system. Next, we provide the most important details about the OWL implementation of our conceptual model in Section 4.2. In Section 4.3, we proceed by discussing the use of Prolog for detecting interactions. Finally, we revisit the audit and confidentiality example conflict in Section 4.4.

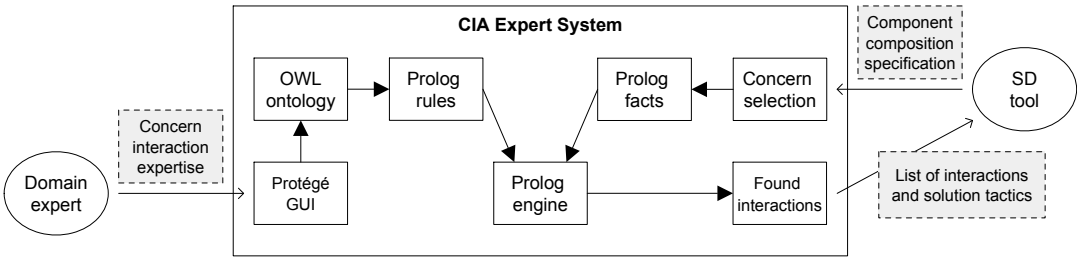

Fig. 6. Architecture of the CIA expert system.

4.1 Overview

The expert system is built upon two technologies. Firstly, OWL is used together with the Protégé [32] environment for representing knowledge about interactions. Secondly, we use Prolog for querying and reasoning about knowledge from the database. Note that OWL also provides some form of reasoning that is limited to class and instance inferences, which is not enough to express our interaction rules. Figure 6 shows how these two technologies are used together. Expertise about interactions between concerns is added to the OWL ontology by domain experts who use the Protégé graphical user interface for this. The OWL representation is generated by Protégé. Subsequently, the acquired interaction knowledge is automatically transformed into a set of Prolog rules. Secondly, a software development (SD) tool has to provide a specification of a certain concern composition to be investigated for potential concern interactions. Essentially, this specification consists of a list of the selected concerns. Based on this list, a set of Prolog facts is generated that contains all the predicate definitions that describe the listed concerns. Both the set of Prolog rules and Prolog facts are fed into a Prolog engine that through reasoning can detect all the interactions that occur in the given concern composition. This list of interactions finally is presented back to the software development tool. We consider the available concern interaction knowledge to be rather stable in time while, on the contrary, concern composition specifications can easily vary greatly for different concern interaction acquisition requests. We now zoom into the use of OWL, Protégé and Prolog. Finally, we illustrate the whole on our running example.

4.2 Ontology-based representation of concern interaction knowledge

To start with, we implemented the conceptual model under the hood of the CIA expert system as an OWL ontology in Protégé, a widely known open source ontology editor and knowledge-base framework. By definition, an ontology is a data model that represents a domain (in our case concern interactions) and can be used to reason about the objects in that domain and the relations between them. Ontologies are commonly used as a form of knowledge representation for a variety of purposes including inductive reasoning, classification, problem solving techniques and to facilitate communication and information sharing. They are generally made up of concepts (classes), relations between these classes and characteristics of individual classes. OWL stands for Web Ontology Language and it is designed for use by applications that need to process the content of information. OWL facilitates greater machine interpretability of content than that supported by XML, RDF, and RDF Schema by providing additional vocabulary along with a formal semantics [28]. Protégé is capable of automatically generating the OWL representation and it also assists the domain experts as it automatically checks the consistency of the inserted knowledge and also automatically completes it with new inferred knowledge.

4.3 Reasoning

We generate a set of Prolog [31] rules based on the OWL implementation of our conceptual model that serves as our concern interaction knowledge base. We wrote a parser in Java using XPath [44] enabling the transformation of the OWL code[3] into a set of Prolog rules. We explicitly did not opted for existing tools that combine OWL and Prolog, which we motivate in Section 5. Next to concern interaction expertise that is expected from domain experts, the CIA system also requires the specification of a given concern composition from a software development tool. The selection of the concerns to be composed for example can be done by drag and dropping them onto a canvas. The list of all concerns within such a specification is deduced from the knowledge database. Based on this list of concerns,

[3] We used RDF as the syntax for presenting the OWL ontology because it is more structured and consistent than Protégé's standard abbreviated OWL syntax and, hence, simplifies the parsing work.

we select the corresponding predicates that describe these concerns out of our ontology. It is exactly this list of predicates that we use as the second input for the Prolog engine. The engine will match the set of facts against the set of Prolog rules. The reasoning then results in the set of interactions that occur within the specified concern composition. Via these interactions, we can again query the OWL ontology for tactics on how to solve the interactions.

4.4 Our example revisited

In order to illustrate the steps that allow us to have a set of Prolog rules, we show the example definition of the conflict between audit and confidentiality by providing some snapshots of its OWL representation that is generated by Protégé in Figure 7. Lines 1 – 6 cover the definition of the interaction with as name `ConflictBetweenAuditAndConfidentiality`. The definition indicates the type of the interaction (line 2) which can be concluded automatically through OWL reasoning based on the relationship of the rule that explains this interaction. This rule is mentioned at line 5 and defined in lines 25 – 31. Lines 3 and 4 reference the concerns that are involved in this interaction. Both, audit and confidentiality, are described respectively in lines 7 – 15 and 16 – 24. Each concern lists the different interactions it is involved in (lines 9 – 10 and 18 – 19), the components it is realised by (lines 11 – 12 and 20 – 21) and the different predicate instances that describe the concern under consideration (lines 13 – 14 and 22 – 23). Notice that the latter ones match with the predicates contained within the description of the rule (lines 29 –30) that explains the original interaction. Finally, lines 32 – 38 give the representation of the audit predicate (the one for confidentiality is similar), indicating the head and the relevant parameter. Our parser will start looking for all known rules that explain an interaction together with their specific relationship and the predicates that are contained within each rule, followed by getting the concerns that are involved in the interaction a rule describes. As a result, the following Prolog code can automatically be generated and matched against the facts based on this OWL code.

```
% ConflictBetweenAuditAndConfidentiality rule
conflicts_with(audit,confidentiality) :-
    audit(_,_,Object,_),
    confidentiality(Object).
% Facts if the audit and confidentiality service are selected
audit(entity,action,object,result).
confidentiality(object).
```

5 Related work

5.1 Interaction modeling

Chung et al. [9] have defined the NFR framework for representing and analyzing non-functional requirements (NFRs). The framework provides a goal-oriented approach for dealing with NFRs and is intended to help developers produce customized solutions by considering characteristics of the particular domain and system being developed. An essential part of their approach is the notion of softgoal interdependency graphs. These are graphs that represent softgoals and their interdependencies. Such a graph maintains a complete record of development decisions and design rationale in a concise graphical form. A softgoal corresponds in a way to our concern concept. In their work, the interdependencies between softgoals can be of various natures: refinements, contributions, operationalizations, correlations etc. Under the umbrella of their notion of correlations, interactions can be modelled. However, they only take into account conflicts. A NFR type catalogue is another artefact in their work which resembles our concern hierarchy a lot.

```
...
(1)  <rdf:Description rdf:about="#ConflictBetweenAuditAndConfidentiality">
(2)    <rdf:type rdf:resource="#Conflict"/>
(3)    <involves rdf:resource="#AuditI"/>
(4)    <involves rdf:resource="#ConfidentialityI"/>
(5)    <is_explained_by rdf:resource="#rule4"/>
(6)  </rdf:Description>
...
(7)  <rdf:Description rdf:about="#AuditI">
(8)    <rdf:type rdf:resource="#Audit"/>
(9)    <is_involved_in rdf:resource="#ConflictBetweenAuditAndConfidentiality"/>
(10)   <is_involved_in rdf:resource="#..."/>
(11)   <is_realised_by rdf:resource="#auditComponent"/>
(12)   <is_realised_by rdf:resource="#..."/>
(13)   <is_described_by rdf:resource="#AuditPred_23"/>
(14)   <is_described_by rdf:resource="#..."/>
(15) </rdf:Description>
...
(16) <rdf:Description rdf:about="#ConfidentialityI">
(17)   <rdf:type rdf:resource="#Confidentiality"/>
(18)   <is_involved_in rdf:resource="#ConflictBetweenAuditAndConfidentiality"/>
(19)   <is_involved_in rdf:resource="#..."/>
(20)   <is_realised_by rdf:resource="#confidentialityComponent"/>
(21)   <is_realised_by rdf:resource="#..."/>
(22)   <is_described_by rdf:resource="#ConfidentialityPred_11"/>
(23)   <is_described_by rdf:resource="#..."/>
(24) </rdf:Description>
...
(25) <rdf:Description rdf:about="#rule4">
(26)   <rdf:type rdf:resource="#Rule"/>
(27)   <explains rdf:resource="#ConflictBetweenAuditAndConfidentiality"/>
(28)   <has_relationship rdf:resource="#conflicts_with"/>
(29)   <contains_predicate rdf:resource="#AuditPred_23"/>
(30)   <contains_predicate rdf:resource="#ConfidentialityPred_11"/>
(31) </rdf:Description>
...
(32) <rdf:Description rdf:about="#AuditPred_23">
(33)   <rdf:type rdf:resource="#AuditPred"/>
(34)   <describes rdf:resource="#AuditI"/>
(35)   <is_part_of_rule rdf:resource="#rule4"/>
(36)   <head rdf:datatype="http://www.w3.org/2001/XMLSchema#string">audit</head>
(37)   <object rdf:datatype="http://www.w3.org/2001/XMLSchema#string">object</object>
(38) </rdf:Description>
...
```

Fig. 7. Part of the OWL representation of a concern interaction

Feature models represent hierarchies of properties of domain concepts [33, 10]. The properties are used to discriminate between concept instances, i.e. systems or applications within that domain. The properties are relevant to end users. At the root of the hierarchy there is the so-called concept feature, representing a whole class of solutions. Below of this concept feature there are hierarchically structured sub-features showing refined properties. Feature models are used for development and application of software product lines, i.e. for defining products and configurations, for describing possibilities of a product line, and for establishing new products and adding new properties to a product line [18]. Compared to our work, a feature model matches more or less with our concern hierarchy. An instance of the feature model corresponds to a selected set of concerns in our approach. In [4], the authors use an algebraic theory for modeling interactions in feature-oriented designs in which feature interactions are modeled as derivatives.

Work on integrating ontologies and rules also exists. Rules are the next layer of the Semantic Web [41] that is the subject of currently ongoing research. Ontologies form the highest layer that is sufficient mature and are a first step from adding reasoning to pure domain descriptions. The combination of both promises to offer enhanced representation and reasoning capabilities. SweetProlog [21] is a system for translating web rules into Prolog enabling an integration of ontologies and rules. This is achieved via a translation of OWL ontologies and OWLRuleML rules into a set of facts and rules in Prolog. Antoniou et al. [2] implemented DR-Prolog, a powerful declarative system supporting rules, facts and ontologies together with all major Semantic Web standards. We deliberately chose not to use these systems, because the rules we need are already modelled through our OWL ontology implementation of the conceptual model and hence, can be easily generated from the ontology itself.

5.2 Interaction detection and resolution

Wohlstadter et al. [42] have presented GlueQoS, a middleware-based approach to managing dynamically changing QoS requirements (quality of service issues related to non-functional requirements such as security, reliability and performance) of components. They use policies to advertise non-functional capabilities. These policies vary at run-time with operating conditions. GlueQoS also incorporates middleware enhancements to match, interpret, and mediate QoS requirements of clients and servers at deployment time and/or runtime. The latter is their main contribution. In their work, they assume a fixed ontology of features, with all interactions explicitly identified ahead of time. The link with our work here is obvious. Moreover, they also provided a classification of feature interactions. The similarities with our categorization makes us believe both interaction classifications are very close to a sweet spot. Another piece of work on conflicts in policy-based distributed systems management is done by Lupu et al. [24].

In the field of aspect-oriented software development [16], aspect interactions represent one of the biggest remaining challenges. In this context, Pawlak et al. [30] propose CompAr, a language that allows programmers to abstractly define an execution domain, advice codes and their often implicit execution constraints. In our opinion, the high level of abstraction the language offers to specify very generic aspect definitions is their major contribution. Moreover, their language enables the automatic detection and solving of aspect-composition issues (interactions between aspects) of around advices. A number of other approaches with respect to automatic detection and resolution of interactions exist [14, 19, 34, 13]. For example, [14] proposes a language-independent technique to detect semantic conflicts among aspects that are superimposed on the same join point. Their approach is based on a resource-operation model. They argue that a formalization of the complete behaviour of a component is not realistic; we agree. However, they don't motivate that their abstraction mechanism is designed in such a way that it is possible to represent the essential behaviour of an aspect. [19] describes interference (i.e. interactions) between aspects at the semantic level, irregardless of any overlap among joinpoints or variables. Their definition of interference resembles our definition of conflicts a lot. They assume that each aspect already has a specification and is correct with respect to that specification. The

specification of an aspect consists of a set of assumptions and guarantees which both are expressed in temporal logic. Based on these specifications, they want to generate proofs that one aspect does not interfere with another one.

6 Conclusion

In this paper, we presented a conceptual model that helps a software development team to understand and manage the different interactions in a component-based distributed application. To demonstrate results, we have focussed on the particular application domain of common middleware services. However, we are convinced that the models and techniques presented here are equally applicable to other domains as well. In general, the conceptual model complements methods for building large and complex component-based distributed systems. We addressed two problems. First, we defined a conceptual model for explicitly representing knowledge about interactions between concerns and how to solve these. As a result, this knowledge can be shared and used in the course of system evolution. Secondly, we provide practical support to the software developer for managing the interactions when he/she is creating, adapting or evolving the component composition of an application. Finally, we discussed our prototype implementation of the CIA expert system that uses a concern composition specification as input for detecting interactions that occur in the given set of concerns.

References

1. G. Booch, and W. Kozaczynski, “Component-Based Software Engineering”, Software, IEEE, vol. 15, no. 5, p. 34 - 36, 1998.
2. G. Antoniou, and A. Bikakis, “DR-Prolog: A System for Defeasible Reasoning with Rules and Ontologies on the Semantic Web”, IEEE Transactions on Knowledge and Data Engineering, vol. 19, no. 2, p. 233-245, 2007.
3. Aspects, Dependencies and Interactions Workshop, ECOOP 2006, Nantes, France, http://www.aosd-europe.net/adi06/.
4. J. Liu, D. Batory, and S. Nedunuri, “Modeling interactions in feature oriented systems”, International Conference on Feature Interactions (ICFI), June 2005.
5. L. Blair, and J. Pang, “Feature interactions - Life beyond traditional telephony”, FIW 2000, p. 83 - 93.
6. L. Blair, G. Blair, and J. Pang, “Feature interaction outside a telecom domain”, Workshop on Feature Interaction in Composed Systems, 2001.
7. M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec, “Feature interaction: A critical review and considered forecast”, Computer Networks: The International Journal of Computer and Telecommunications Networking archive, Vol. 41 , Issue 1, p. 115-141, January 2003.
8. CAM/DAOP, Component-Aspect Model / Dynamic Aspect-Oriented Platform, http://caosd.lcc.uma.es/CAM-DAOP/index.htm.
9. L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos, “Non-Functional Requirements in Software Engineering”, Kluwer Academic Publishing, 2000.
10. K. Czarnecki, and U. W. Eisenecker, “Generative Programming”, Addison Wesley, 2000.
11. A. Dardenne, A. Van Lamsweerde, and S. Fickas, “Goal-directed Requirements Acquisition”, Science of Computer Programming, vol. 20, p. 3 - 50, 1993.
12. J. A. Diaz Pace, F. Trilnik, and M. R. Campo, “How to handle interacting concerns?”, Workshop on Advanced for Separation of Concerns in OO Systems, OOPSLA 2000, Minneapolis, USA.
13. R. Douence, P. Fradet, and M. Sudholt, “Composition, reuse and interaction analysis of stateful aspects”, International Conference on Aspect-Oriented Software Development (AOSD04), 2004.
14. P. Durr, L. Bergmans, M. Aksit, “Reasoning about Semantic Conflicts between Aspects”, Proceedings of Aspect, Dependencies, and Interactions (ADI) Workshop, 2006.
15. Enterprise JavaBeans Technology, http://java.sun.com/products/ejb/white_paper.html, 1998.
16. R. Filman, T. Elrad, S. Clarke, M. Aksit, “Aspect-oriented software development”, Addison-Wesley, 2004.

17. R. J. Hall, “Feature interactions in electronic mail”, Proceedings of the 6th International Workshop on Feature Interactions in Telecommunications and Software Systems, IOS Press, 2000.
18. K. C. Kang, K. Lee, and J. Lee, “FOPLE - Feature Oriented Product Line Software Engineering: Principles and Guidelines”, Pohang University of Science and Technology, 2002.
19. S. Katz, “Aspect categories and classes of temporal properties”, Transactions on Aspect Oriented Software Development (TAOSD), LNCS 3880, p. 106 - 134, 2006.
20. D. O. Keck, and P. J. Kuehn, “The feature and service interaction problem in telecommunications systems: A survey”, IEEE Transactions on Software Engineering, vol. 24, no. 10, October 1998.
21. L. Laera, V. Tamma, T. Bench-Capon, and G. Semeraro, “SweetProlog: A system to integrate ontologies and rules”, In Rules and Rule Markup Languages for the Semantic Web, Proceedings of the 3rd RuleML workshop, LNCS 3323, p. 188 - 193.
22. T. Lindholm, and F. Yellin, “The Java Virtual Machine Specification”, Addison-Wesley, Reading, MA, 1997.
23. X. Liu, G. Huang, W. Zhang, and H. Mei, “Feature interaction problems in middleware services”, International Conference on Feature Interactions (ICFI), June 2005.
24. E. Lupu, and M. Sloman, “Conflicts in policy-based distributed systems management”, IEEE Transactions on Software Engineering, vol. 25 , Issue 6, p. 852-869, November 1999.
25. B. Meyer, “Object-oriented software construction (2nd ed.)”, Prentice-Hall, Inc., 1997.
26. Object Management Group, “The Common Object Request Broker: Architecture and Specification Revision 2.4”, OMG Technical Document, 2000.
27. Object Management Group, “CORBAservices: Common Object Service Specification”, OMG Technical Document, 1998.
28. OWL Web Ontology Language, Overview, http://www.w3.org/TR/owl-features/.
29. D.L. Parnas, “On the Criteria to be Used in Decomposing Systems into Modules”, Communication of the ACM, vol. 15, no. 12, 1972.
30. R. Pawlak, L. Duchien, and L. Seinturier, “CompAr: Ensuring safe around advice composition”, 7th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS05), Athens, Greece, June 2005.
31. SWI-Prolog’s Home, http://www.swi-prolog.org/.
32. The Protégé Ontology Editor and Knowledge Acquisition System, http://protege.stanford.edu/.
33. M. Riebisch, “Towards a More Precise Definition of Feature Models”, in M. Riebisch, J. O. Coplien, D, Streitferdt (Eds.): Modelling Variability for Object-Oriented Product Lines, BookOnDemand Publ. Co., Norderstedt, 2003.
34. M. Rinard, A. Salcianu, and S. Bugrara, “A classification system and analysis for AO programs”, Proceedings of the Twelfth International Symposium on the Foundations of Software Engineering, Newport Beach, CA, November 2004.
35. A. Sampaio, R. Chitchyan, A. Rashid, and P. Rayson, “EA-Miner: a tool for automating aspect-oriented requirements identification”, Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering (ASE), 2005.
36. F. Sanen, E. Truyen, W. Joosen, A. Jackson, A. Nedos, S. Clarke, N. Loughran, and A. Rashid, “Classifying and documenting aspect interactions”, Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (Y. Coady, D. Lorenz, O. Spinczyk, and E. Wohlstadter, eds.), pp. 23-26, Bonn, Germany, 2006.
37. F. Sanen, E. Truyen, W. Joosen, N. Loughran, A. Rashid, A. Jackson, A. Nedos, and S. Clarke, “Study on interaction issues”, AOSD-Europe Deliverable 44, 2006, http://www.aosd-europe.net/deliverables/d44.pdf.
38. R. Schantz, and D. C. Schmidt, “Middleware for Distributed Systems”, Encyclopedia of Computer Science and Engineering, edited by Benjamin Wah, 2007.
39. D. Schmidt, and S. Huston, “C++ Network Programming: Resolving Complexity with ACE and Patterns”, Addison-Wesley, Reading, MA, 2001.
40. T. Thai, and H. Lam, “.NET Framework Essentials"’, O’Reilly, 2001.
41. W3C Symantic Web Activity, http://www.w3.org/2001/sw/.
42. E. Wohlstadter, S. Tai, T. Mikalsen, I. Rouvellou, and P. Devanbu, “GlueQoS: Middleware to Sweeten Quality-of-Service Policy Interactions”, In Proc. of the International Conference of Software Engineering, 2004.
43. A. Wollrath, R. Riggs, and J. Waldo, “A Distributed Object Model for the Java System”, USENIX Computing Systems, 1996.
44. XML Path Language, http://www.w3.org/TR/xpath.