Augmenting UML Models for Composition Conflict Analysis

Andreas Leicher and Jörn Guy Süß

Technische Universität Berlin, Germany,
Computergestützte InformationsSysteme (CIS)
{aleicher, jgsuess}@cs.tu-berlin.de

Abstract. Component reuse is inhibited by two factors: Lack of an adequate modeling representation of components and lack of a method to predict properties of a composition of application components. In this paper, we propose a framework for conflict identification. The framework is primarily based on a taxonomy describing communication and technology related properties. Conflict identification is based on inference rules. Furthermore, we aim to integrate conflict reasoning in the software development process. We will show that the Unified Modeling Language and the Resource Description Framework can be combined to provide a solution to the representation problems, without resorting to extension mechanisms, and without limiting to a specific component platform. As a real life example, we model the connection of an .Net Serviced Component to an Enterprise Java Bean as part of a mortgage bank's enterprise architecture and prove its viability.

1 Introduction

The advantages of buying a fitting component to provide a part of a solution over custom construction are long established [7]. The number of components we can consider to build a solution depends on the size of the market [14] from which we can buy the solution. Unfortunately, platform boundaries subdivide this market, because we can not evaluate if components of different technologies are compatible or not.

Each technology can be described by properties relevant for communication. We aim to support conflict analysis for middleware components based on such properties. We reuse an adapted version of the connector taxonomy proposed by Medvidovic/Mehta as this taxonomy provides more fine grained properties compared to other approaches. As this taxonomy is designed platform independently, we customized the taxonomy for particular middleware technologies. Based on this taxonomy, developers can estimate on the fly, how complicated a particular composition will be.

Furthermore, we aim to support component analysis in the context of the software design process. Design is often based on abstract models, that are represented by diagrams in a graphical notation like that of the UML. We perceive a method to quickly estimate component compatibility in the context of these tools valuable, because

M. Cerioli (Ed.): FASE 2005, LNCS 3442, pp. 127–140, 2005.

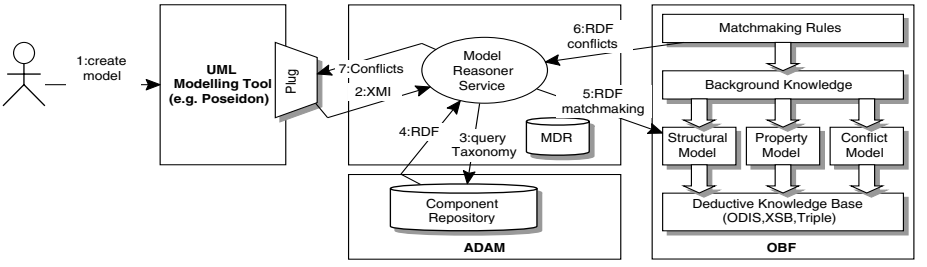

Fig. 1. Architecture of the Ontology-Based Framework

– developers can decide on the fly how complex a composition is.
– analysis can be carried out in the normal development process, without the need to transfer data into a specialized analysis tool.

While the metamodel of the UML is designed to accommodate object-oriented languages, it does not offer straightforward support for deductive logic, which we would need to draw conclusions about the compatibility of components.

We propose to attach only the necessary information about components to elements in a model, and then reason about this information externally against our domain-specific background knowledge. Our approach provides the ability to check that this additional meta-information fulfills structural constraints like type specifications, and thus guarantees the validity of input to services based on such information.

Figure 1 shows an overview of the overall process. Execution proceeds as follows: Within a UML tool - in this case Poseidon UML - we create a component model[1]. Components are annotated with the property information available. This includes properties describing the technologies as well as other properties that are known to the developer (see figure 2 for an example). We connect the components with an association and attach a comment which indicates to the service that this association is to be processed (1). Then we submit the model to the Model Reasoner Service embedded in EVE[2] [22] (2). The service extracts the annotations in the model and attaches itself to a repository designed to hold Analytical Data on Architectures and Models (ADAM). The service extracts the addressed part of the knowledge base (3/4) and passes it to the reasoner, combined with the information extracted from the model(5). The reasoner calculates the match and returns its characteristics to the service(6). The service embeds the resulting information in the model, attached to the association (7). If the result is a conflict, a conflict description is generated. If the result is a match, the service can fill in implied property information for each component, if desired by the user.

[1] In principle, this can be done either with UML 1.x or 2.0. However, most existing tools support only UML 1.x so that we use a profile to describe components.
[2] EVE is a framework to support tool independent manipulation of UML models.

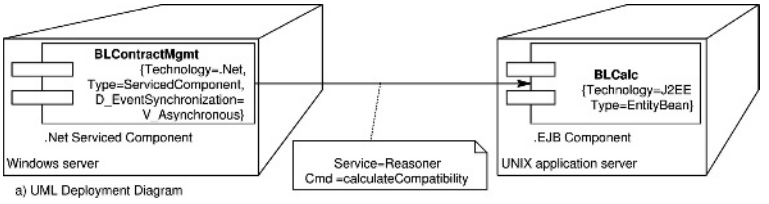

Fig. 2. Integrating two heterogeneous components, initial step

Example component annotations can be found in figure 2. Both components are part of a customer information system of a large European mortgage bank: the simulation of financial development for different forms of mortgage contracts, predicting expected savings for combinations of financial products. The system consists of two components, a management component which acts as a customer facade and a set of worker components, which provide the calculation. The calculation functionality ('BLCalc') is implemented in a piece of Java code originally developed for a standalone application. To share the functionality the code was encapsulated as an EntityBean component and deployed on a UNIX application server. The 'BLContractMgmt' Component was implemented as a .Net Serviced Component on a windows server.

The rest of the paper is structured as follows: In section 2 we discuss our platform independent framework for reasoning about component matches and its schema. Section 3 discusses the state of the art regarding background knowledge in UML diagrams and introduces our RDF-based solution. Section 4 summarizes the approach and widens the discussion to include other fields of application.

2 Conflict Identification for Component Composition

In this section, we first provide an overview of existing approaches to classify component-based systems as well as of approaches to identify conflicts. We then introduce our framework for conflict identification, explain the compatibility relationship used to identify conflicts and discuss some results obtained by analyzing our running example.

2.1 Existing Approaches

Architectural Styles were one of the early approaches in software architecture to classify systems. A style specifies the parts of a system as well as properties that need to be satisfied in a system configuration. Bass et. al. [2] define an architectural style as follows:

> By a particular style we mean a set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem, together with local or global constraints on the way composition is done.

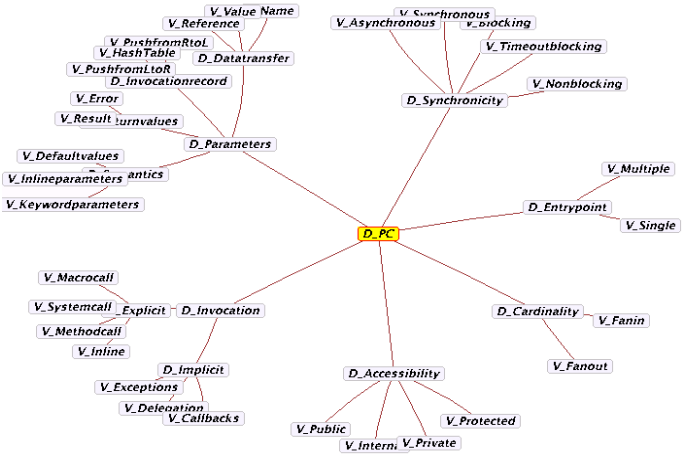

Fig. 3. Procedure Call: Starting from the Connector Type Procedure Call, including Dimensions and Values as defined by Mehta

One possible application area of styles is the classification of systems regarding the composition of their constituting parts. Shaw [19, 20] provides such a classification. A style is represented by a set of values that describe the kinds of components and connectors, the control and data flow as well as their interactions in a system. Another application area infers resulting properties of a particular style. A style, for example, is indicative of such aspects as system reconfiguration, component exchange, and component adaptation.

In general, an architectural style takes a kind of 'macro' view of a system. It describes 'coarse grained' properties that must hold for a whole system. These are helpful, if we investigate the system as a whole. However, only a few of these properties are relevant for deciding compatibility of single components. Unfortunately, the classification provided by Architectural Styles is not useful for analyzing middleware technologies such as CORBA, J2EE etc. These systems show almost no differences in the classification. Middleware systems aim for similar goals and are designed with similar architecture in mind.

Medvidovic/Mehta [15, 17, 16] propose a sophisticated taxonomy to describe communication properties of connectors. Part of this taxonomy, rendered by our ODIS tool [5] is shown in figure 3. It describes relevant properties for a procedure call. This taxonomy consists of eight connector types, each of which is described by several dimensions (complex properties) that consist of subdimensions and values. Each connector can provide four kinds of services: communication, coordination, conversion, and facilitation. Connectors in programming languages and middleware technologies can be described by deriving and extending the taxonomy.

The analysis of middleware technologies in the context of Mehta's connector taxonomy also reveals several problems: The terms used in the taxonomy are not explicitly defined. For a number of terms, a lot of different definitions are available. Some terms are ambiguous as different interpretations in the taxonomy can be chosen. For example Exceptions can refer to a method that throws

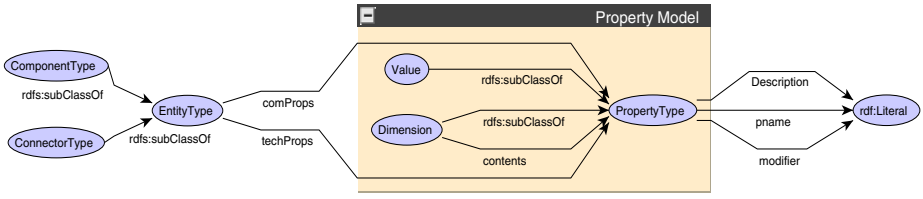

Fig. 4. The Property Model of the Framework

an exception or one that is activated because of an exception. The taxonomy describes connector types as part of the taxonomy. We feel that these types should not be included in the taxonomy, because connector types are the entities that are described by properties, but they themselves are not properties.

Other approaches aim to automatically discover mismatches based on conflicting characteristics [10, 9, 1, 12, 24, 16]. Most of these approaches concentrate on architectural mismatches. They do not handle technologies directly as well as structural and behavioral specifications. They can be classified in approaches using only a structure such as a table to describe properties [1, 16] and in approaches which additionally provide reasoning support [10, 9, 12]. Approaches providing reasoning support often only support a subset of properties available in the former category.

2.2 Property Model

To analyse the relationships of communication and technology related properties, we need to define a means of notation. Our property model shown in figure 4 defines a structure to create hierarchically connected properties. Each property is described as a feature that can either be optional or mandatory. A feature can contain several sub-features. Sub-features can be grouped by two operators 'xor' and 'or' to describe possible feature combinations. Furthermore, each feature can be associated with attributes (FeatureAttributes) to state additional requirements. 'EntityTypes' can be associated with 'Features' by two relationships: one to describe communication properties (comProps) and one to state technology related properties (techProps).

To organize the space of component properties, we decided to reuse the existing taxonomy by Medvidovic/Mehta, as it provides the most fine grained properties. Unfortunately, this taxonomy is designed on a platform independent level. Therefore, we needed to analyse platform specific connectors for middleware systems of interest. We modified the original taxonomy in the following way:

- Platform specific properties that describe communication in Java, Jini, J2EE and .Net were added.
- A modifier for conflict analysis was introduced. It describes whether a property is 'mandatory' or 'optional' in a given context.
- The meaning of properties was exactly defined: A definition is associated with every property.

- Connector types were removed from the taxonomy.
- Name clashes that occur due to the removed connector types were resolved.

A second taxonomy covers the aforementioned technological properties such as platforms (OSs), programming languages, etc. As we have not found any existing taxonomy that covers these properties, we have defined them from scratch. Technology-related information, such as the language, in which a component is written, platform availability or resulting cost provide additional information regarding the complexity of a connector. For example, it may describe if a composition of two components requires a distributed connector, or if they can be composed by a local connector. We do not detail this taxonomy in this paper, as it does not relevantly contribute to the topic at hand.

2.3 The Role of Communication Properties Regarding Composition

The communication taxonomy describes properties in the context of connector types. However, for our example, we also need to interpret communication properties in the context of component types.

If we represent each technology (EJB, ServicedComponents) by a middleware component, i.e. a binary artifact, each application component (BLCalc, BLContractMgmt) is bound via a precisely defined mechanism to that middleware and cannot be used independently. Figure 5 shows a typical mechanism that uses stub and skeleton objects to integrate components with respect to a particular middleware. Here, middleware plays a dual role: It is at the same time a connector that facilitates the communication and a component that can be physically deployed in an appropriate location.

Consequently, application components (BLCalc, BLContractMgmt) are restricted by the properties offered by their technologies. As we have described communication properties of EntityBeans and ServicedComponents [11], we are able to annotate application components (BLCalc, BLContractMgmt).

Regarding a composition, components are either the initiators of a communication or the receiver. Conflict identification needs to only consider the relevant properties for such a constellation. For example, an EntityBean is annotated with properties concerning data access to a underlying database. In a communication where the EntityBean is called by a client, however, these properties need not to be considered, because the client is not concerned with database issues.

Figure 6 shows the example components annotated with communication properties relevant for communication initiated by the 'BLContractMgmt' component.

2.4 Conflict Identification

We assume that the connector taxonomy as well as the taxonomy for technology related properties contain all relevant properties for communication. Component comparison is based on the comparison of annotated properties based on their type (either optional or mandatory). Two entities are compatible, if for all vertices they either require a property or do not support it. For example, two

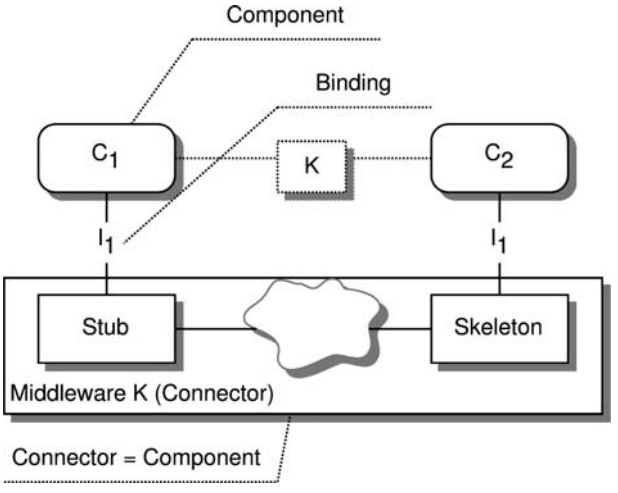

Fig. 5. Component Binding in Subject to the Underlying Middleware

components C_1 and C_2 are compatible, if the predicate $comp(C_1, C_2)$ evaluates to true:

$$\begin{aligned} &\forall n \in \{C_1.r.comProps \cup C_1.r.techProps\} \\ &\quad isMandatoryFeature(n) \rightarrow \\ &\quad \exists m \in \{C_2.q.comProps \cup C_2.q.techProps\} \cdot \\ &\quad n.fname = m.fname \wedge \\ &\quad isMandatoryFeature(m) \end{aligned} \tag{1}$$

Unfortunately, this approach suffers several problems:

1. Often, it is difficult to decide if a property is required in a particular technology or not. Middleware specifications describe communication protocols coarse grained only. Lower level properties are often not or only partially described. Consequently, we need to deal with unknown properties.
2. Technologies support several communication mechanisms that may be used by application components, but are not compulsory. Consequently, we must distinguish between a property that is supported as an option or that is required (mandatory).

In response to these problems each feature can be described by one the following states:

Optional: The property can be supported by the component.
Unsupported: The component does not support this property.
Mandatory: The component requires this property for communication.

As a result, we get a compatibility matrix (shown in table 1) describing valid and invalid property combinations between the two components to be composed. We

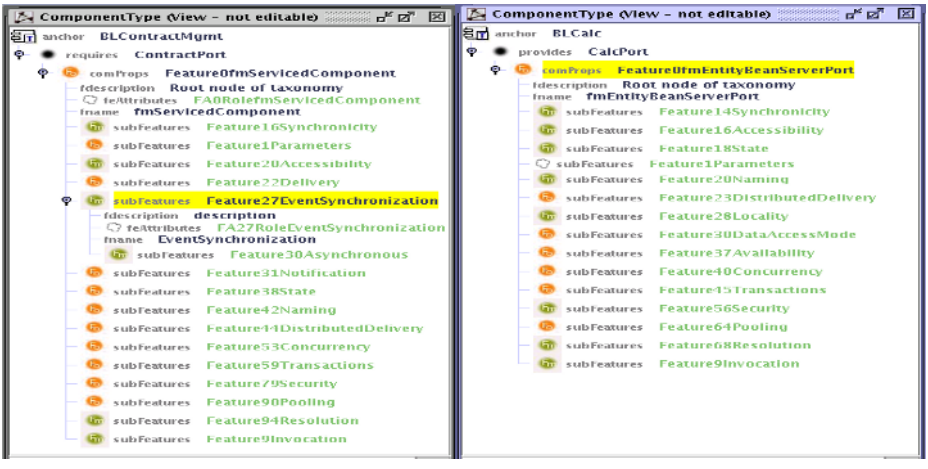

Fig. 6. View of the Communication Properties of the Example Components in Their Middleware Context

Table 1. Compatibility Matrix between two Components

Component vs. Component	mandatory	optional	unsupported
mandatory	√	w	f
optional	w	w	w
unsupported	f	w	√
w = warning f = failure			

distinguish conflicts of two categories: Failures are generated if properties definitely do not match, e.g. 'unsupported' vs. 'mandatory'. Warnings are generated due to 'optional' properties. For example, if 'BLCalc' is annotated with an 'optional' property it is unclear if it actually supports the property or not. Consequently, a warning needs to be generated. Optional properties are often annotated to connector types and middleware components such as the EntityBean component type. They describe the communication mechanism provided by a technology. These mechanisms may be used by application components but are not required.

To express conflict rules we require a logical formalism. As the component descriptions can be viewed as instances of a more general component schema, a formalism like F-Logic [13], which distinguishes between instance data and schema information (types/classes) would be advantageous.

Triple [21] satisfies these conditions. It is a language designed for reasoning in the semantic web. Triple states facts as quadruples (S,P,O,C): S for subject, the entity to be described. P is a predicate that states the relation of interest, O stands for an Object, which is either a Literal or another quadruple. C describes

the context within which the tuple is valid. Thus, Triple facts are RDF statements extended by the 'context', which allows specifying views of an object in different contexts. This feature is extremely helpful, because it divides up fact bases into chunks that can be used as separate units.

In addition Triple provides two further advantages: It allows universal identification of resources through introduction of URIs. Section 3 shows how this can be applied. Also Triple allows the creation of new contexts on the fly by definition of mapping rules. We have applied this to transform a UML model containing platform independent components into different EJB component realizations. The concrete transformation is implicitly selected by requirements stated as parameters to the transformation rule[6].

Conflicts are generated by 'Triple' rules of the following kind:

```
forall ?c,?s,?f,?n,?pc,?ps unsupportedMandatoryFeatures(?c,?pc,?s,?ps,?f)<-
  getComFeatures(?c,?pc,?f) and
  hasOnlyMandatoryParentFeatures(?f) and
  getFeatureName(?f,?n) and
  isFeatureNotBound(?s,?ps,?n).

forall C,S,PC,PS @failure(C,S,PC,PS) {
  forall ?x, ?f, ?ns
    ?ns:?x[sys:directType->core:FeatureConflict;
       core:concerns->C;
       core:relates->S;
       core:concernsFeature->?ns:?f;
       core:cause->'Mandatory feature of client unsupported by server.']
    <-
       unsupportedMandatoryFeatures(C,PC,S,PS,?ns:?f)@core and
       concatConflict(?x,?f,'Failure').
}
```

The first rule identifies mismatched properties. The second rule (a mapping) generates conflict statements. These statements can be directly converted to plain RDF and handed back to the modeling tool.

2.5 Conflicts in the Example

Analysis of the example components yields several conflicts, part of them shown in figure 7. For example, a failure concerns Event support (Feature30Asynchronous). An EntityBean cannot handle events. These are covered by MessageBeans in J2EE. Furthermore, naming schemes (Feature21Structurebased, Feature22Hierarchical) are handled differently in both technologies: J2EE uses a structure based naming mechanism, .Net an attribute based scheme. Furthermore, a lot of warnings (not shown) are generated, because most property values are imported directly from the underlying component types (EntityBean and ServicedComponent). As discussed above these components are often 'optional'. So, it cannot be inferred that they actually match.

As shown in figure 1 conflicts are handed back to the EVE Service (6). The results are attached to the association (7) and presented to the developer. Conflicts

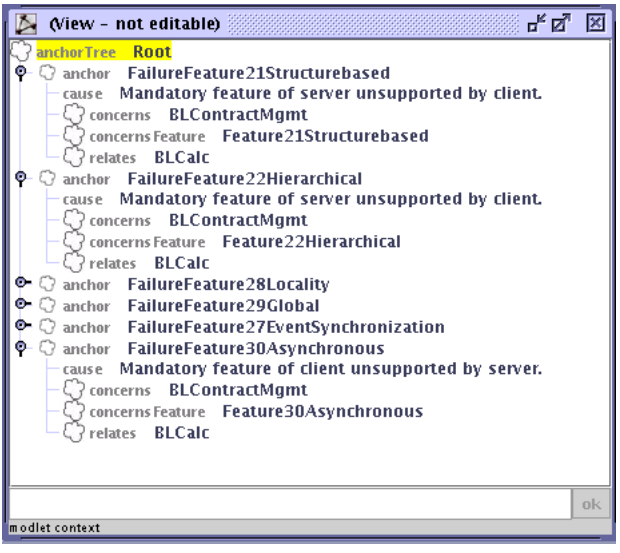

Fig. 7. Conflicts Generated by Comparing Both Components

need to be interpreted by a developer. She needs to select important properties and prune superfluous properties. In any case, it should be possible for a developer to infer, what is actually needed for composition and how this impacts on cost and resources.

3 Augmenting the UML with an Overlaid RDF Structure

In UML 1.x, external resources like files can only be represented as components and artifacts, which can only be used in component and deployment diagrams.To make statements about such external resources, an association is drawn from the element which is assigned the property to the artifact which represents the associated resource. These associations rarely appear in diagrams, because they cut across diagram types, making their practical application difficult. In addition, the choice of component types available is limited and the extension of that type space involves the creation of UML profiles. If only one profile is allowed under the version of UML in use, the modeler has to choose to either apply the profile she uses for her primary problem domain and drop detailed modeling of the types of background resources, or model the background resources in detail and drop the profile for the domain, or manually create a unified profile, which may lead to clashes between stereotype constraints. In any case, all the information which one merely wanted to attach has to be included in the model in a tedious way, because it involves substantial indirection: Information about the external resource is linked to a Model Element that is created solely to act as a placeholder for that type of metadata.

Even in today's UML 2.0, there is no simple mechanism to attach background knowledge to the model. Some case tools like Rational Rose work around this limitation by introducing links to other resources as a new type of Model Element residing in Packages next to other Model Elements. These link-based extensions, apart from being proprietary to the tool, have several disadvantages:

- The meaning of the resource that the link points to is only weakly defined. For example, if a link, which points to an HTML page about a Java library, resides in package "x", implications are unclear. It could mean that the library realizes package "x" or that the model relies on package "x" or that the author of this model used patterns described on the page to create the contents of that package.
- Links are also limited because they can only be directly attached to Packages and no other Model Elements. For example, to state that a Class acted in the role of a ConcreteCommand in the Command Pattern, one would need to create a link in the Package and additionally describe the relationship to that link on a Comment attached to the Class. Furthermore, such a Comment may be ignored when interpreting the model elsewhere, since a Comment "has no semantic force but may contain information useful to the modeler." [18–pp.2-28]

While the previously described extensions of plain UML are either tedious and problematic, like the creation of profiles for resource description, or unprecise like the use of unqualified package links, the extension mechanism offers the Tagged Value - a useful yet simple feature[18–pp. 2-68]: "An arbitrary property attached to the Model Element. The tag is the name of the property and the value is an arbitrary value. The interpretation of the Tagged Value is outside the scope of the UML metamodel." However, Tagged Values do not have any descriptive power with regards to outside resources. But this can be introduced by defining a convention describing how to link and type such resources. In fact, the definition of Tagged Values can be interpreted similarly to the representation of knowledge about resources in the Resource Description Framework (RDF) [23–3.11]: "The underlying structure of any expression in RDF is a collection of triples, each consisting of a subject, a predicate and an object. A set of such triples is called an RDF graph ... Each triple represents a statement of a relationship between the things denoted by the nodes that it links." So to join Tagged Values and RDF we only need a bijective function, which maps Tagged Values to RDF statements and vice versa.

In a nutshell, our approach involves the following steps: Assign a Tagged Value to each Model Element to be annotated. Choose its Name to be the Uniform Resource Identifier (URI)[3] of the properties' definition and the Value to be the URL of the resource. Extract the RDF. Query or reason in logic.

3.1 RDF Statements

The example in listing 1.1 shows how the .Net Calculation component is described as supporting asynchronous communication by linking it with an external resource via a semantically well-defined relationship. This relationship is expressed by an RDF triple which describes a Tagged Value. The parts of the

triple are the equivalent of a sentence with subject, predicate and object. The grammatical elements are:

Subject. There is a Component named "BLContractMgmt" which resides in the UML model in a Package named "Business Apps".

Predicate. It can be described in terms of event synchronization support, as defined in the core of the .Net Serviced Component Taxonomy of the CIS group.

Object. From the different choices on event synchronization, it does only support asynchronous message transfer.

Listing 1.1. An RDF description of a component's transaction capability

```
1 (S) <.#Business%20Apps::BLContractMgmt>
2 (P) <http://cis.cs.tu-berlin.de/picm/core/
      ServicedComponent#EventSynchronization>
3 (O) <http://cis.cs.tu-berlin.de/picm/core/
      ServicedComponent#Asynchronous> .
```

Listing 1.1 might require some explanation: The statement is written in a simple RDF-equivalent notation called N3 [4]. Each element is a URI. Primarily these serve to identify resources for the purpose of retrieval. In that role they function as a Uniform Resource Locator (URL) as can be seen in line one: The subject of the description, which is the "BLContractMgmt" component, can be accessed by navigating into the Namespace element called "Business Apps" to the Model Element called "BLContractMgmt" within the current document.

Thus, to use this approach to describe an organization's own concepts it has to decide on the types of properties it would like to apply to its models. Then URIs, keyed off of the organizations Internet domain name, can be assigned to represent the desired properties[3].

3.2 Extraction of RDF Statements from Tagged Values

The extraction service follows this algorithm: Extract each model element, and see if any Tagged Values exist. If so, convert its name into a URI as follows: Take as root the relative or absolute URL, describing the location of the model file. Append as path the model-internal path based on the Namespace of the Model Element. Append the name of the Model Element. This results in unique URIs because of the scoping of Model Elements within Namespaces[18–pp. 2-38]: "The pathname of Namespace or ModelElement names starting from the root package provides a unique designation for every ModelElement." The model element forms the subject of the RDF statement. The predicate is the name of the associated tagged value. The object is the value of the tagged value. As a result, RDF statements as defined in section 3.1 are created.

[3] Please note that a URI must not link to a web representation. It is a concept used to uniquely identify resources.

Our implementation of extraction is a JMI-based service which uses the Jena[8] framework. We call that service 'Fringe' because it extracts information pointing from the fringes of the UML model to external resources, rather than at structures within the model.

4 Conclusion

This paper has discussed a framework based on a connector taxonomy to enable an adequate modeling representation of components and provide a method to discover conflicts in compositions of those components. The framework is based on a taxonomy by Mehta. It describes communication and technology related properties and provides conflict identification based on inference rules. We have discussed how to integrate such conflict reasoning into the software development process. To this end, we have shown how the Unified Modeling Language and the Resource Description Framework can be combined via Tagged Values to provide a solution to representation problems, without resorting to extension mechanisms, and without limiting to a specific component platform. As a real life example, we have modeled the connection of a .Net Serviced Component to an Enterprise Java Bean in UML and identified inherent conflicts using the Triple-based framework.

We plan to augment the conflict reasoning framework to suggest solutions to conflicts based on existing connectors that are registered in our knowledge base. Furthermore, type checks and behavior checks are to be integrated with the matchmaking process.

References

1. Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(3):213–249, 1997.
2. Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Software Engineering Institute. Addison-Wesley, 1998. ISBN 0-201-19930-0.
3. T. Berners-Lee, R. Fielding, and L. Masinter. RFC 2396: Uniform Resource Identifiers (URI): Generic syntax, 1998. Status: DRAFT STANDARD.
4. Tim Berners-Lee. Notation 3 - ideas about web architecture, 2001. `http://www.w3.org/DesignIssues/Notation3.html`.
5. Andreas Billig. ODIS - Ein Domänenrepository auf der Basis von Semantic Web Technologien. In *Tagungsband der Berliner XML Tage*. XML-Clearinghouse, 2003. english version: http://www.isst.fhg.de/~abillig/Odis/xsw2003.
6. Andreas Billig, Susanne Busse, Andreas Leicher, and Jrn Guy S. Platform independent model transformation based on triple. Middleware 2004, October 2004.
7. Jr. Frederic P. Brooks. *The Mythical Man Month*. ISBN: 0-201-83595-9, anniversary edition, 1995.
8. Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: Implementing the semantic web recommendations. Technical Report HPL-2003-146, Hewlett Packard Laboratories, 24 2003.

9. L. Davis, D. Flagg, R. Gamble, and C. Karatas. Classifying interoperability conflicts. In T. Weng H. Erdogmus, editor, *COTS-Based Software Systems: Second International Conference, ICCBSS 2003 Ottawa, Canada*, number 2580 in LNCS, pages 62–71. SPRINGER, 2003.
10. L. Davis, R. Gamble, and J. Payton. The impact of component architectures on interoperability. *Journal of Systems and Software*, 61(1):31–45, 2002. based on the Technical Report UTULSA-MCS-99-30.
11. Jan Gädicke. Metadatengestützte analyse der kommunikationsfähigkeit von enterprise java beans und .net. Master's thesis, TU Berlin, 2004. german.
12. A. Kelkar and R.F. Gamble. Understanding the architectural characteristics behind middleware choices. In *1st International Conference in Information Reuse and Integration, 1999.*, 1999.
13. M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42:741–843, Juli 1995.
14. M.D. McIlroy. Mass produced software components. In P.Naur and B. Randel, editors, *NATO Conference on Software Engineering*. NATO Science Commitee, Oktober 1968.
15. Nikunj R. Mehta. Software connectors: A taxonomy approach. In *Workshop on Evaluating Software Architectural Solutions 2000*. Institute for Software Research University of California, Irvine, 2000.
16. Nikunj R. Mehta and Nenad Medvidovic. Understanding software connector compatibilites using a connector taxonomy. In *In Proceedings of First Workshop on Software Design and Architecture (SoDA02), Bangalore, India*, December 2002.
17. Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22nd international conference on Software engineering*, pages 178–187, 2000.
18. Object Management Group (OMG). *Unified Modeling Language Specification, Version 1.3*, March 2000. `http://cgi.omg.org/docs/formal/00-03-01.pdf`.
19. Mary Shaw and Paul C. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *Proceedings of the 21st International Computer Software and Applications Conference*, pages 6–13. IEEE Computer Society, 1997.
20. Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. PH, April 1996. ISBN 0131829572.
21. Michael Sintek and Stefan Decker. TRIPLE - A Query, Inference, and Transformation Language for the Semantic Web. In *Proceedings of International Semantic Web Conference ISWC 2002*. Lecture Notes in Computer Science, Bd. 2342, Springer, 2002.
22. Jörn Guy Süß, Andreas Leicher, Herbert Weber, and Ralf-D. Kutsche. Model-centric engineering with the evolution and validation environment. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language: Modeling Languages and Applications, 6th International Conference, San Francisco, CA, USA*, volume 2863 of *LNCS*, pages 31 – 43. Springer, 2003.
23. World Wide Web Consortium. *Resource Description Framework (RDF) Model and Syntax Specification*, 1999. statut : W3C Recommandation, errata REC-rdf-syntax-19990222 , http://www.w3.org/TR/REC-rdf-syntax/.
24. Daniil Yakimovich, James M. Bieman, and Victor R. Basili. Software architecture classification for estimating the cost of cots integration. In *Proceedings of the 21st international conference on Software engineering*, pages 296–302. IEEE Computer Society Press, 1999.