JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2008

Vol. 8, No. 7, November–December 2009

Components, Contracts and Vocabularies - Making Dynamic Component Assemblies more Predictable

Jens Dietrich, Graham Jenson, Massey University, School of Engineering and Advanced Technology (SEAT), Palmerston North, New Zealand, Email: j.b.dietrich@massey.ac.nz,grahamjenson@maori.geek.nz

In recent years, dynamic component-based systems such as OSGi and its derivatives have become very successful. This has created new challenges for verification. Assemblies are created and modified dynamically at runtime, but many existing techniques such as unit testing are designed for buildtime verification. Runtime verification is usually restricted to type checks. We propose a simple component contract language that is powerful enough to represent different types of complex contracts between collaborating components, including contracts with respect to component semantics and quality of service attributes, and contracts that refer to resources other than programming language artefacts. These contracts are based on a pluggable contract vocabulary and can then be used for runtime verification of assemblies. We present a proof of concept implementation of the contract language proposed for the OSGi/Eclipse component model.

1 INTRODUCTION

Component-based systems have become very popular in the last decade. While initially used mostly in desktop application, component-based software engineering is now used in many different areas including server-side and ubiquitous computing. This has created a number of new challenges for component models with respect to component lifecycle and resource management. Traditionally, component models focus on one particular aspect in order to describe the relationship between collaborating components - interface compatibility. This relationship is defined by a contract that is usually expressed by means of programming language artefacts like Java interfaces, or by using a dedicated interface definition language (IDL). However, modern component models have to address use cases where other types of contracts are involved. For instance, server applications often require a high level of reliability, and applications running on mobile devices have special requirements with respect to the (hardware) resources components can use. If components are dynamically discovered, it might not be enough to know that these components provide the right interface, they must also have the expected behaviour. Beugnards et al. [BJPW99] have investigated types of component contracts and have classified contracts into four layers:

1. Basic syntactic contracts expressing interface compatibility.

2. Behavioural contracts specifying component semantics.

3. Synchronisation contracts describing dependencies between components.

4. Quality of service contracts describing requirements with respect to response times, quality of results etc.

Beugnards et al. have also discussed several technologies that could be used to express contracts for the various layers. This includes the use of IDLs for layer 1, design by contract [Mey92] for layer 2 and TAO [SLM98] for level 4. Other types of contracts not covered by this classification include aspects related to security, trust and licensing. For instance, an organisation might want to prevent the use of components with contagious licenses, or configurations where components with incompatible licenses are linked together. In OSGi, version 4.2 of the specification will contain features that make licensing information part of the machine readable component meta-data and will enable applications to reason about this (RFC 125, [OSGb]).

In some modern component frameworks even basic layer 1 contracts can be rather complex. A good example is the successful component model used by Eclipse [Ecl]. Based on OSGi [OSGa] bundles, Eclipse plugins use extension points and extensions to define required and provided resources. Often, these resources are Java types - plugins define extension points using Java interfaces, and require other plugins providing extensions to these extension points to supply classes implementing the respective interfaces. However, in general these contracts are highly polymorphic. An example for this is the `org.eclipse.help.toc` extension point. In order to extend it, applications have to provide help resources and a table of content XML file instantiating a given document type definition. Moreover, many extension points use complex logical expressions. An example for this is `org.eclipse.ui.actionSets`. Here, the value of the attribute `class` must be a name of a class that implements an interface. Which interface this is depends on the value of another attribute (`style`).

In this paper, we introduce Treaty, a component contract language designed to address these issues. The paper is organised as follows: In section 2, we introduce the Treaty contract language. We use an Eclipse-based example application for this purpose. This application contains polymorphic and disjunctive contracts, and uses unit test cases for layer 2 and layer 4 contracts. We then discuss contract instantiation and verification. In section 4 we show how contract vocabularies can be organised in a modular manner. In section 5 we explore the use of unit test cases in contracts in more detail. We then discuss the architectural aspects of Treaty, focusing on the relationship between contracts and the underlying component model. A discussion of related work and open questions concludes our contribution.

The Treaty framework and the example used throughout this paper are both accessible on Google code[1], the code is licensed under the Apache open source license.

2 FORMALISING CONTRACTS

Components collaborate in different ways. When designing component-based systems in an object-oriented language, the most common way of collaboration is that one component provides an abstract type, while another component provides (an instance of) an implementation class of this abstract type. The use of abstract types decouples the collaborating components. As mentioned in the introduction, modern component-based systems like Eclipse also use different types of contracts. For instance, components have to supply XML documents instantiating document type definitions (DTDs) or XML Schemas. In general, we can consider the artefacts provided and consumed by components as resources identified by uniform resource identifiers (URIs). These resources are typed, examples for types are instantiable Java classes, Java interfaces, IDL interfaces, XML instances, XML Schemas, XSL files, DTDs, property files, and CSV files. Relationships associating resources are defined for certain resource types only, for instance Java classes implement Java interfaces, XML documents instantiate DTDs, or style sheet transformations applying to instances of a certain XML Schema.

In [DHG07] it has been proposed to use the semantic web standards RDF, OWL and SWRL to model component contracts in a platform-independent manner. The resource description framework (RDF) [KC04] provides a generic standard to describe resources. In addition, the web ontology language (OWL) [MvH04] provides language constructs to define resource types and their relationships, including subclassing and associations between resources of a certain type (object properties). The formal semantics of OWL supports ontology reasoning. Finally, the Semantic Web Rule Language (SWRL) [HPSB$^+$04] supports the definition of derivation rules to define predicates. In [DHG07] it has been shown how contracts can be expressed using SWRL rules defining an "extends" predicate. While this has some obvious advantages, including the existence of a formal semantics for SWRL, the resulting rules are too complex and do not support a compact representation of contracts. Furthermore, these contracts have restricted expressiveness. In particular, complex constraints using exclusive disjunctions cannot be represented.

For this reason, we have developed a custom XML vocabulary that supports the compact definition of component contracts. This vocabulary is part of Treaty, the contract framework we propose. Contracts define relationships between two parties: consumer and supplier. Treaty as a framework abstracts from the concrete nature of these entities. For the example used here we use the proof of concept implementation

[1]http://code.google.com/p/treaty/, the Eclipse update site URL is http://treaty.googlecode.com/svn/trunk/treaty-eclipse-updatesite/site.xml. The Treaty plugin requires JDK 1.6 or better.

of Treaty for the Eclipse component model. Here, the consumer and supplier roles are mapped to extension points and extensions of Eclipse plugins.

Figure 1 shows such a contract [2]. The respective example is implemented as a set of Eclipse plugins. In this contract, the relationship between a component that prints dates (clock view) and a component that provides a date formatting service (date to string) is defined. The contract is attached to the Eclipse component that has the extension point as an XML file in the component meta-data folder (`META-INF`). The name of this file is defined by the following naming convention: the name of the extension point followed by the extension `.contract`. This mechanism is non-invasive - contracts can be added to plugins without modifying existing plugin resources. Treaty does not modify the Eclipse plugin registry either - it is only queried through public interfaces and if there are no contracts found for an extension point it is interpreted as empty contract.

A Treaty contract has three parts:

1. In the consumer section (lines 3-19), the resources of the extension point are defined. The resources defined are constants identified by name and type. The types are defined in an (external) ontology and represented by URIs. This information can be used by the component to load resources if needed, for instance by using the component class loader.

2. In the supplier section (lines 20-27), the resources of the extension are defined. This is where a component provides resources to be consumed by a consumer. These resources are also typed. Resources are now variables, the `ref` element is used to define a variable that can be used to query for the resource once a concrete extension is known. This reflects the support for dynamic component models that use late binding. Details of this mechanism are discussed further below.

3. In the constraints part (lines 28-45), the relationships between resources of both sides are specified. The schema supports the use of standard logical connectives such as AND, OR and XOR to define complex conditions. In addition to relationships, value properties and existence conditions are supported as well.

In the example shown in figure 1, the clock component that has the extension point provides the following resources (package names for classes omitted):

1. The interface `DateFormatter` (id “`Interface`”) that describes the interface of the date formatter service.

2. The `dateformat.xsd` (id “`DateFormatDef`”) schema that describes the interface of an alternative service by means of an XML schema. Instances of this schema define date formatting string templates.

[2]The package names are abbreviated

3. The class `DateFormatterFunctionalTests` (id "`FunctionalTests`") defines some JUnit functional test cases. The test cases check whether the strings produced by a date formatter contain at least the day, the month (as number or using the English name of the month) and the last two digits of the year. They define the minimal information content of strings rendering dates.

4. The class `DateFormatterPerformanceTests` (id "`QoSTests`") defines JUnit quality of service tests. It checks whether a date formatter needs less than 10ms to render a date.

The extending component must provide one of two resources: a Java class or an XML document. The contract conditions state that a valid extension must either provide an XML instance that is valid with respect to the schema, or an instantiable class that implements the interface and passes the additional functional and performance tests.

Conditions in contracts can be either atomic or complex. To build complex conditions, the usual logical connectives with their standard semantics can be used - see table 1 for details. Three types of atomic conditions are supported: relationships between resources, resource properties, and conditions that a resource must exist. Relationships and properties are equivalent to object and data properties in RDF. The `mustExist` constraint is weaker - this merely asserts that the respective resource must exist and must be of the declared type. Table 2 specifies the attributes of the atomic contract conditions. The attribute types used are the XMLSchema built-in types defined in [XST].

condition type	semantics
property	comparison of a property of a resource with a literal using a comparison operator
relationship	establishes whether a typed relationship exists between two resources
mustExist	true iff the referenced resource exists
not	true iff the contained condition is false
or	true iff at least one of the contained conditions is true
xor	true iff exactly one of the contained conditions is true
and	true iff all of the contained conditions are true

Table 1: Contract condition types and their semantics

3 CONTRACT LIFECYCLE AND VERIFICATION

The sample contract is still abstract since it references resources (the resources of the supplier) that are not yet known at the time the contract is written. The supplier is only known later at runtime when late binding occurs. Only then the contract

```xml
<?xml version="1.0" encoding="UTF-8"?>
<contract>
  <consumer>
    <resource id="Interface">
      <type>http://www.treaty.org/java#AbstractType</type>
      <name>clock.DateFormatter</name>
    </resource>
    <resource id="QoSTests">
      <type>http://www.treaty.org/junit#TestCase</type>
      <name>clock.DateFormatterPerformanceTests</name>
    </resource>
    <resource id="FunctionalTests">
      <type>http://www.treaty.org/junit#TestCase</type>
      <name>clock.DateFormatterFunctionalTests</name>
    </resource>
    <resource id="DateFormatDef">
      <type>http://www.treaty.org/xml#XMLSchema</type>
      <name>/dateformat.xsd</name></resource>
  </consumer>
  <supplier>
    <resource id="Formatter">
      <type>http://www.treaty.org/java#InstantiableClass</type>
      <ref>/serviceprovider/@class</ref></resource>
    <resource id="FormatString">
      <type>http://www.treaty.org/xml#XMLInstance</type>
      <ref>/serviceprovider/@formatdef</ref></resource>
  </supplier>
  <constraints>
    <xor>
      <and>
        <relationship
          resource1="Formatter" resource2="Interface"
          type="http://www.treaty.org/java#implements"/>
        <relationship
          resource1="Formatter" resource2="FunctionalTests"
          type="http://www.treaty.org/junit#verifies"/>
        <relationship
          resource1="Formatter" resource2="QoSTests"
          type="http://www.treaty.org/junit#verifies"/>
      </and>
      <relationship
        resource1="FormatString" resource2="DateFormatDef"
        type="http://www.treaty.org/xml#instantiates"/>
    </xor>
  </constraints>
</contract>
```

Figure 1: XML Contract Example

condition	property	type	semantics
property	type	URI	the type of the property, the type usually refers to an XSD built-in datatype
	resource	IDREF	reference to the resource that has the property
	value	anySimpleType	a value literal
	operator	URI	a comparison operator, usually one of the XPath/XQuery built-in operators
	property	URI	a property of the respective resource
relationship	resource1	IDREF	a reference to the source
	resource2	IDREF	a reference to the target
	type	URI	the type of the relationship
mustExist	resource	IDREF	the resource that must exist

Table 2: Properties of contract conditions

can be instantiated. Contract instantiation is the creation of a deep copy of the contract, and the instantiation of all resource proxies in this copy. A resource proxy is a resource that has a `ref` attribute but no `name` attribute. The `ref` attribute is a reference to the components meta-data. The Treaty framework contains an interface `ResourceManager` that is used to resolve those proxies. The details of resolving are component-model specific. In the Eclipse-based implementation of Treaty, the `ref` values are XPath expressions and the `ResourceManager` uses them to query the plugin meta-data (`plugin.xml`). In an implementation of Treaty for pure OSGi the attribute values could just be simple strings representing keys of properties defined in the bundle manifests.

Once an instantiated contract exists, verification can be performed. This is the checking of conditions according to their semantics. When complex conditions are present, this is usually done using a top-down strategy. This is a simple process: once all resources are instantiated, contracts are essentially statements of classical propositional logic, and usually they are not very complex. The interesting question is how the basic conditions are checked. This requires the resources to be loaded. For instance, to check properties of a resource of the type `AbstractType`, the respective class must be loaded so that it can be analysed using the Java reflection framework. This is done with a `ResourceLoader`. Again, on the framework level this is an interface that must be implemented when adapting Treaty for a particular component model. In case of Eclipse, the loader uses the OSGi bundle classloader to load resources. Figure 2 shows the different resource states and the respective transitions.

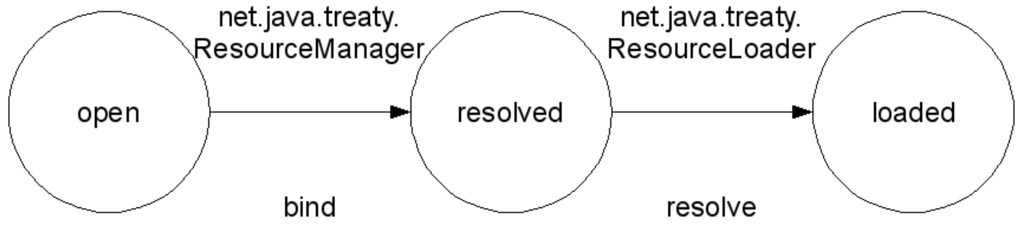

Figure 2: Resource states

4 CONTRACT VOCABULARIES

Contracts reference types and properties. Both can be formally defined in a formal ontology language, but this alone does not define their semantics [Usc01]. For instance, the semantics of the (Java) `implements` predicate (figure 1, line 33) is the set of pairs of concrete Java classes C and Java interfaces I such that C implements I. In other terms, the semantics can be defined by a function that takes two resources C and I and can compute a boolean indicating whether $(C, I) \in implements$ is the case or not. This particular function is easy to provide: if C and I can be loaded and are available as instances of `java.lang.Class`, the method `isAssignableFrom` can be used to check this condition. In a similar manner, a validating XML parser can be used to check the `instantiates` property associating XML instances and XML schemas.

A possible solution to this problem is to define a fixed type system that contains a set of commonly used resource types, including some code that represent the semantics of the respective properties and relationships. However, there might be very project-specific types and relationships to be used in contracts. Consider a scenario where a company has a product with a reporting extension point. This offers customers the option to plug-in their own reporting templates with customised layouts and data aggregation. The resource type to be provided by these components could be `VelocityTemplate`[Vel]. Or, even better, a product-specific `MyReportTemplate` type that represents velocity templates that use only a fixed set of variables which the host component can bind. Then the component itself would make contributions to the contract vocabulary in order to enable verification. There is a clear business use case for this: it safeguards the company against faulty third party plugins which would result in customers blaming the company or other plugin providers for the malfunctioning of their software.

Therefore, the vocabulary should be kept open and extensible. This can be achieved by using the component model itself to build modular contract vocabularies. Each vocabulary component must provide the following:

1. A list of defined types (URIs) contributed by the component.
2. A list of defined properties (URIs) contributed by the component.
3. A list of defined relationships (URIs) contributed by the component.
4. A method to load a resource given a reference and a resource type. For instance, this method is used to load resources of the type Java class defined by an attribute in `plugin.xml` as Java classes using the plugin's class loader.
5. A method that can be used to check the properties and relationships contributed by the component.

In the Treaty implementation for Eclipse, this functionality is defined through the extension point `net.java.treaty.eclipse.vocabulary`. To extend this extension point, plugins must implement an interface that has the methods to load resources and check conditions, and have to provide an OWL resource that defines the vocabulary extensions. Figure 4 shows the contract for the vocabulary extension point, figure 3 shows a code fragments from the Java class that provides the semantics of the XML vocabulary. This class is an implementation of the `net.java.treaty.ContractVocabulary` interface. In this class, the `xml#instantiates` relationship is checked by using a validating XML parser.

Treaty merges the ontology contributions into a central merged ontology. This ontology contains all contributed types, properties and relationships, plus annotation indicating which component contributed the respective artifacts. Figure 5 depicts the merged ontology that is used by Treaty for Eclipse with on-board vocabulary contributions only. The graph is visualised using the W3C style, in addition, shaded areas represent the components that contribute the respective vocabulary elements. In the RDF graph, this information is represented using annotations. Italic type is used to highlight resources representing object properties.

The reporting template example also shows the benefit of using formal ontologies. For instance, assume that the reporting template type `MyReportTemplate` subclasses `VelocityTemplate`, and that the contract requires only the existence of a reporting template. Using the semantics of `rdfs:subClassOf`, the verifier could then first check whether the resource is of the type `VelocityTemplate` by using the Velocity parser. If this fails, the resource cannot be an instance of `MyReportTemplate` either. That is, the formal semantics of OWL can be used to optimise verification. Another ontology feature that is useful here is are `rdfs:subPropertyOf` relationships between properties and relationships.

For this reason, in the proof of concept implementation all components making vocabulary contributions have access to a central singleton `Vocabulary` that maintains the virtual merged ontology. This allows them to use ontology reasoning when

```java
import java.io.InputStream;
import java.net.URL;
import javax.xml.XMLConstants;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.xml.validation.*;
import net.java.treaty.*;
public class XMLVocabulary implements ContractVocabulary {
...
public void check(RelationshipCondition condition)
  throws VerificationException {
  String rel = condition.getRelationship().toString();
  if ("http://www.treaty.org/xml#instantiates".equals(rel)) {
    try {
      URL schemaURL = (URL)condition.getResource2().getValue();
      URL instanceURL = (URL)condition.getResource1().getValue();
      SchemaFactory factory =
        SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
      Schema schema = factory.newSchema(schemaURL);
      Validator validator = schema.newValidator();
      InputStream in = instanceURL.openStream();
      Source source = new StreamSource(in);
      validator.validate(source);
      in.close();
    } catch (Exception x) {
      throw new VerificationException("validation failed" ,x);
    }
  }
  else
    throw new VerificationException("predicate not supported");
}
...
}
```

Figure 3: Class providing the semantics for relationhips contributed by the XML vocabulary (simplified)

```
<?xml version="1.0" encoding="UTF-8"?>
<contract>
  <consumer>
    <resource id="VocabularyDef">
      <type>http://www.treaty.org/java#AbstractType</type>
      <name>net.java.treaty.ContractVocabulary</name>
    </resource>
  </consumer>
  <supplier>
    <resource id="ContributedVocabulary">
      <type>http://www.treaty.org/java#InstantiableClass</type>
      <ref>/vocabulary/@class</ref>
    </resource>
    <resource id="Ontology">
      <type>http://www.treaty.org/owl#Ontology</type>
      <ref>/vocabulary/@ontology</ref>
    </resource>
  </supplier>
  <constraints>
    <and>
      <relationship
        resource1="ContributedVocabulary" resource2="VocabularyDef"
        type="http://www.treaty.org/java#implements"/>
      <mustExist resource="Ontology"/>
    </and>
  </constraints>
</contract>
```

Figure 4: Contract for vocabulary extensions

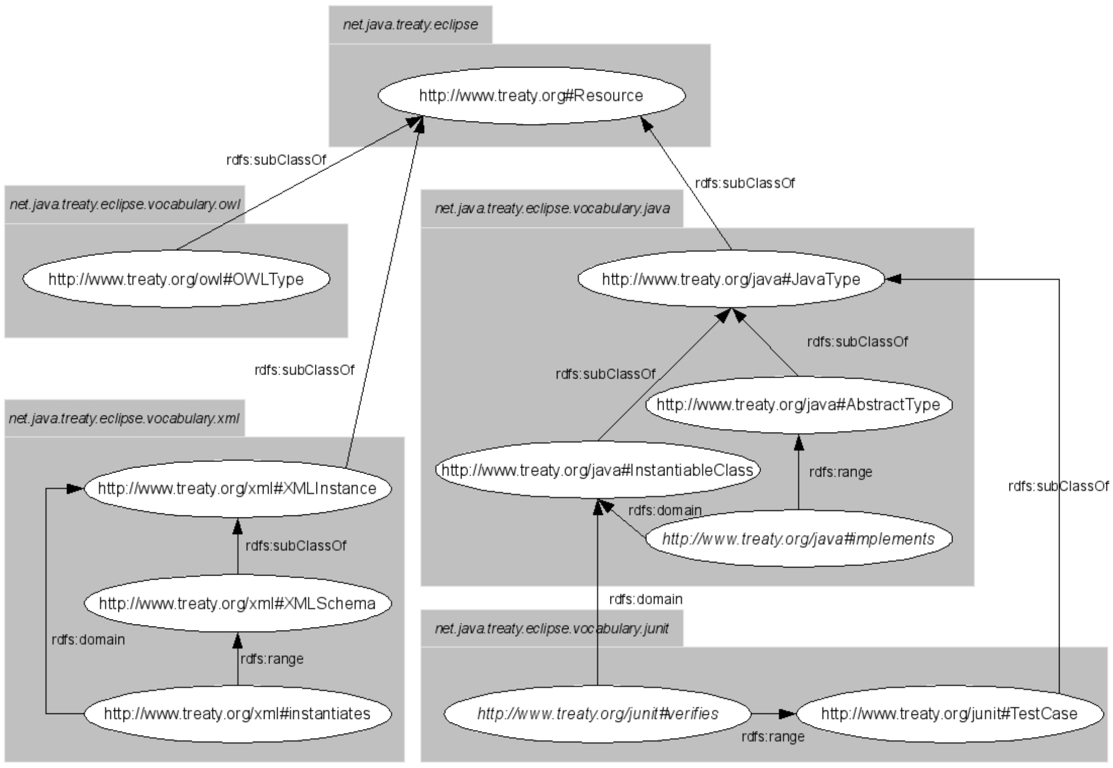
net.java.treaty.eclipse
http://www.treaty.org#Resource
rdfs:subClassOf
net.java.treaty.eclipse.vocabulary.owl
http://www.treaty.org/owl#OWLType
rdfs:subClassOf
net.java.treaty.eclipse.vocabulary.java
http://www.treaty.org/java#JavaType
rdfs:subClassOf
rdfs:subClassOf
http://www.treaty.org/java#AbstractType
http://www.treaty.org/java#InstantiableClass
rdfs:range
rdfs:domain
http://www.treaty.org/java#implements
rdfs:subClassOf
net.java.treaty.eclipse.vocabulary.xml
http://www.treaty.org/xml#XMLInstance
rdfs:subClassOf
http://www.treaty.org/xml#XMLSchema
rdfs:domain
rdfs:range
http://www.treaty.org/xml#instantiates
rdfs:domain
net.java.treaty.eclipse.vocabulary.junit
http://www.treaty.org/junit#verifies
rdfs:range
http://www.treaty.org/junit#TestCase

Figure 5: Merged contract vocabulary used in the example program

checking contributed properties and relationships. The ontology can be accessed as unparsed stream or as instance of `org.semanticweb.owl.model.OWLOntology` [BVL03].

5 UNIT TESTING AT RUNTIME

The example contract (figure 1) uses the `verifies` property to express minimum requirements with respect to functionality and performance for classes implementing the `DateFormatter` interface. This relationship is based on JUnit, that is, the test resources are JUnit 4 test cases, and the semantics of the relationship is defined by means of a JUnit test runner. JUnit test cases are defined in the same component that defines the date formatter interface. These tests check whether date formatter implementations can convert dates in less than 10ms, and whether the generated strings contain at least tokens representing date, month and year.

Unit testing is particularly useful here as it stands in the tradition of design by contract - describing the semantics of methods through a description of the state changing effects of the methods expressed by pre- and post conditions. The main weakness of unit testing when compared to other verification methods is that verification is based on selected specimen objects. Tests are not sufficient to prove or ensure correctness, they can only be used to approximate it. The main advantage of unit tests is that they are widely acceptance by programmers. Also, it is easy to assess the degree of approximation (coverage metrics), and there are well-established development processes to improve test cases when it is necessary to improve the approximation.

Unfortunately, JUnit has been built for design and build time verification. As a consequence of this it is assumed that the classes to be tested are known when the test cases are written and can be directly referenced by test cases. On the other hand, our approach supports late binding at composition time, that is, test cases can only reference abstract types and the actual objects have to be injected if the respective classes become available at runtime. Therefore, JUnit needs to be modified to fit into Treaty. More precisely, support for dependency injection mechanism must be added to JUnit. This is achieved by designing test cases that have constructors with parameters that can be used to inject the tested objects before the test case life cycle starts, and a special test runner that can instantiate test cases using this constructor. Such a test runner is part of the Treaty component that makes the JUnit vocabulary contributions.

6 THE BIGGER PICTURE - ADDING CONTRACTS TO COMPONENT MODELS

Treaty is implemented in Java and provides support for contract definition and verification for the Java-based Eclipse component model. However, Treaty is largely independent of the underlying component model and could also be used to describe contracts in other component models even if they are not Java-based. In this respect, Treaty is more similar to a scripting language such as JavaScript or a composition language as proposed in [LSNA97]. Treaty itself can be seen as a combination of three separate subsystems:

1. The Contract Definition Language (CDL), a formal language used to define contracts in a platform-independent manner. In this paper we have proposed to use XML (constraint by the treaty.xsd schema) for this purpose, a possible alternative is the use of a general-purpose rule language such as SWRL.
2. The Contract Execution Environment (CEE), a system that reads contracts defined in the CDL and can instantiate and verify the contracts against components of a host component model. The CEE proposed here is implemented in Java and consists of two parts - an abstract contract framework and an implementation of the abstract concepts in the framework for the OSGi/Eclipse component model.
3. The Contract Vocabulary (CV), an ontology that defines the types and properties that are used in contracts.

Figure 6 shows the subsystems, their internal relationships and the relationships to the host component model (CM). In particular, the CEE must reference the CM to instantiate resource references using the reflective features of the CM (such as access to meta-data). It also uses the CM to load resources needed to verify constraints. Finally, the CEE provides the semantics for the (data and object) properties used in the vocabulary. The CEE has access to the merged ontology (see figure 5) and can use it for ontology reasoning.

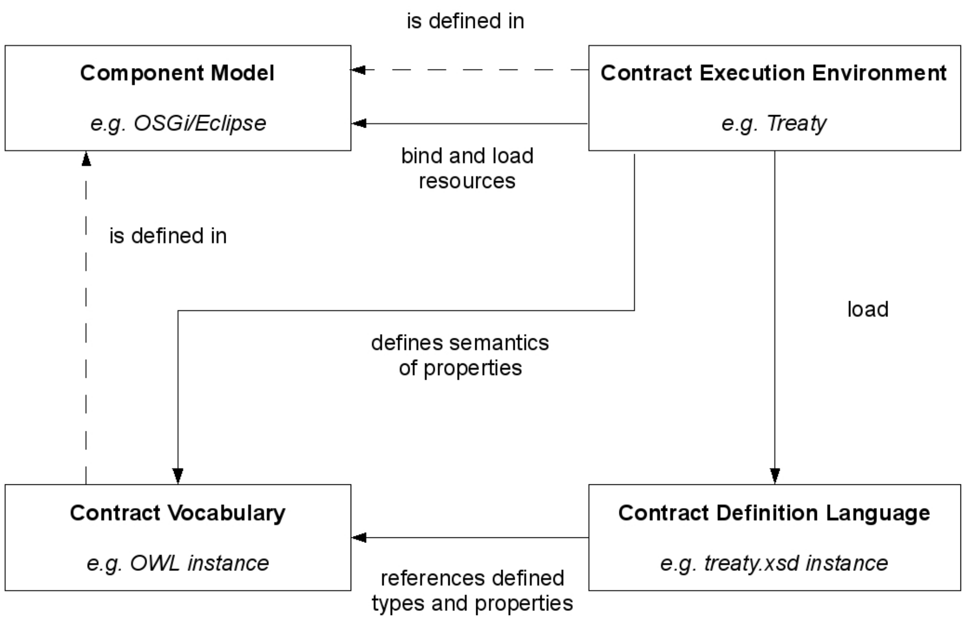

Figure 6: Contract languages and component models

Our Eclipse-based implementation adds two more relationships: both the CV and the CEE take advantage of the CM to define both the vocabulary and the parts of the CEE providing the semantics for the vocabulary in a modular fashion. In figure 6, these relationships are represented by dashed lines.

7 DISCUSSION

We have presented Treaty, a component framework that supports the easy definition of complex and polymorphic contracts. Our main contribution is the contract language, and the modular design of the contract vocabulary. We believe that using such a language adds value to environments that use late binding, such as ubiquitous or mobile computing applications where new components are discovered and integrated at runtime. The types of requirements that need to be expressed in environments like this are somehow unpredictable. We therefore think that using any fixed contract language is not appropriate. Instead, what is needed in an extensible contract language based on a platform-independent description of resource types and their relationships. This allows components to plugin vocabulary extensions that can then be used by verification tools.

There are several related proposals to integrate explicit contracts into component models. Palladio[BKR07] is a performance prediction model. It's main goal is quality of service predictions of architectures based on design models. During the early design stages components are conceived and contracts are defined between components that determine how they perform. These contracts are defined at different stages of the development cycle from different domains such as a component developer and system architect. These contracts are domain specific and provide the basis when testing the overall system performance. Palladio contracts are restricted to layer 1 and 4 contract types. SOFA[PBJ98] is a hierarchical component model designed to provide a platform for software components, coupled with DCUP it supports dynamic evolution of components. SOFA version 2.0[BHP06] is the most recent version and adds additional support for contracts. The component model initially supported interface contracts, behaviour contracts with validation during development, and deployment contracts through connectors. Sofa 2.0 has added the formal modelling of non-functional aspects and greater behaviour validation [Kof07]. The Fractal component model [BCL+06], like SOFA, is a hierarchical component model which mainly focuses on dynamic reconfiguration of complex software systems. Dynamic reconfiguration is achieved through introspection, the discovery of the internal application structure, and intercession. This reconfiguration to be safe verification of the component systems consistency is needed, for this problem a contracting system was developed, ConFract [CRCR05]. This system supports the annotation of interfaces with pre- and post-conditions, and the checking of these conditions at runtime.

Treaty is still rather simple as simplicity was one of the major design goals. One reason for Treaty's simplicity is the fact that much of the work is delegated to the vocabulary contributions. However, in many cases it is rather easy to write these contributions, and the level of reuse for vocabulary elements would be much higher than the level of reuse of the actual application components. The main advantage is that such an open framework supports a consistent representation of different contract types by using a common meta-model (OWL). To the best of our knowledge, no existing (academic or industrial) component models or architectural description language achieves this.

In the prototype we have presented, verification is used as a central service that checks the integrity of the entire system. It might be more useful in many circumstances to check only contracts between certain components, for instance in response to lifecycle events such as component activation or upgrade. As verification also needs resources such as classes that must be loaded and instances that must be created, mechanisms to switch it on and off or contract prioritising (runtime levels) might be useful. Integrating verification closer with the component lifecycle also creates new challenges. In the Eclipse based implementation, verification triggers class loading. This of course is unwanted in a framework like Eclipse that is based on lazy initialisation. There are several possible strategies to deal with with problem, including reseting the plugins after verification, or running verification only if the system is in check or safe mode.

An interesting issue is whether contracts should be attached to components consuming resources (as we have done this), to components providing resources or should be detached from either ("contracts as entities in the middle", as proposed in [Szy00]. On the framework level, Treaty does support contracts on both sides and in the middle, and the aggregation of multiple contracts. The proof of concept implementation based on Eclipse however only support contracts on the consumer side at the moment.

An obvious limitation of Treaty is that verification is strict. That is, verification either succeeds or fails. Sometimes it would be more desirable to have soft verification that computes a value indicating to what extent a component assembly satisfies the contract. We believe that is would be possible to add support for soft verification to Treaty by annotating the contract conditions with ratings, and to use Zadeh operators [Zad65] to compute ratings for complex constraints.

REFERENCES

[BCL$^+$06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quma, and Jean-Bernard Stefani. The fractal component model and its support in java. *Software: Practice and Experience*, 36:1257–1284, 2006.

[BHP06] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, page 4048, Washington, DC, USA, 2006. IEEE Computer Society.

[BJPW99] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–45, 1999.

[BKR07] S. Becker, H. Koziolek, and R. Reussner. Model-based performance prediction with the palladio component model. In *Proceedings of the 6th international workshop on Software and performance*, pages 54–65. ACM Press New York, NY, USA, 2007.

[BVL03] Sean Bechhofer, Raphael Volz, and Phillip Lord. Cooking the semantic web with the owl api. In *In Proc. 2nd International Semantic Web*

Conference (ISWC), Sanibel Island (FL US, pages 659–675. Springer, 2003.

[CRCR05] Philippe Collet, Roger Rousseau, Thierry Coupaye, and Nicolas Rivierre. *A Contracting System for Hierarchical Components*, pages 187–202. 2005.

[DHG07] Jens Dietrich, John Hosking, and Jonathan Giles. A Formal Contract Language for Plugin-based Software Engineering. In *Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 175–184, Washington, DC, 2007. IEEE Computer Society.

[Ecl] Eclipse. http://www.eclipse.org.

[HPSB+04] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. W3C member submission, W3C, May 2004. http://www.w3.org/Submission/SWRL/.

[KC04] Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

[Kof07] J Kofron. *Behavior Protocols Extensions*. PhD thesis, Charles University Prague, 2007.

[LSNA97] Markus Lumpe, Jean-Guy Schneider, Oscar Nierstrasz, and Franz Achermann. Towards a formal composition language. In *Proceedings of ESEC 97 Workshop on Foundations of Component-Based Systems*, pages 178–187, 1997.

[Mey92] Bertrand Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.

[MvH04] Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language Overview. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-owl-features-20040210/.

[OSGa] The OSGi Alliance. http://www.osgi.org.

[OSGb] OSGi service platform release 4 version 4.2 - early draft. http://www.osgi.org/download/osgi-4.2-early-draft.pdf.

[PBJ98] F. Plasil, D. Balek, and R. Janecek. Sofa/dcup: architecture for component trading and dynamic updating. *Configurable Distributed Systems, 1998. Proceedings. Fourth International Conference on*, 0:43–51, May 1998.

[SLM98] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The design of the TAO real-time object request broker. *Computer Communications*, 21(4):294–324, 10 April 1998.

[Szy00] Clemens Szyperski. Components and contracts. Dr Dobbs, May 2000. http://www.ddj.com/architect/184414613.

[Usc01] Michael Uschold. Where is the Semantics in the Semantic Web? In *Workshop on Ontologies in Agent Systems*, Montreal Canada, May 2001.

[Vel] The Apache Velocity Project. http://velocity.apache.org/.

[XST] XML Schema Part 2: Datatypes Second Edition, W3C Recommendation 28 October 2004. http://www.w3.org/TR/xmlschema-2/.

[Zad65] Lotfi Zadeh. Fuzzy sets. *Information and Control*, (8):338–353, 1965.

Jens Dietrich is a senior lecturer at Massey University in New Zealand. His research interests are pattern formalisation, rule engines and component models. He can be reached at j.b.dietrich@massey.ac.nz. See also hhttp://www-ist.massey.ac.nz/jbdietrich.

Graham Jenson is studying towards a PhD in Software Engineering at Massey University, New Zealand. His project is about supporting dynamic evolution with component repository systems. He can be reached at grahamjenson@maori.geek.nz. See also http://maori.geek.nz.