An Ontological Model for Component Collaboration

Jens Dietrich

Massey University
School of Engineering and Advanced Technology
Palmerston North, New Zealand
Email: j.b.dietrich@massey.ac.nz

We propose an ontology-based approach to model dynamic component systems. Component collaborations are described as the provision and consumption of typed resources. We are particularly interested in how the vocabularies used to describe component requirements and capabilities can be managed and used for the verification of assemblies. We discuss how this approach can be applied to improve real-world, industry-strength component models such as OSGi and Eclipse.

ACM Classification: D.1.5 Object-oriented Programming; D.2.2 Design Tools and Techniques; D.2.4 Software/Program Verification; D.2.13 Reusable Software

INTRODUCTION

Building large enterprise systems has always been a difficult, expensive and risky business. The main challenge is the need to deal with ongoing change. This includes both changing requirements and changes in technology. Many innovations in software engineering have been a direct response to these challenges. An example is the emerging of object-oriented software engineering to address the complexity of building graphical user interfaces, and the use of virtual machines to safeguard the deployment of programs in networked environments. Nowadays, software engineering faces new challenges that call for new approaches to produce, manage and use software artefacts. There are multiple social, business and technical factors that drive developments.

Firstly, there is a trend towards software ecosystems. In a software ecosystem, multiple solutions exist that can satisfy the same or similar requirements. Ecosystems create competition amongst suppliers and therefore promote quality and lower expenses. Software ecosystems have been pioneered by the open source community, in particular through platforms such as sourceforge.net and search sites and directories such as freshmeat.net. The best-known examples of products that have emerged from ecosystems are Linux distributions. Only recently, software vendors started to adapt this concept and are now trying to create ecosystems around their products. In some areas like platforms for mobile devices, the existence of such ecosystems is seen as one of the most critical factors for commercial success. IBM has been particularly successful in this regard with the ecosystem that has been developing around the Eclipse development environment. One of the challenges software engineering faces today is how to build software for ecosystems. The main problem here is to find out how to build and publish software so that reasoning about its capabilities and requirements, and about the equality and similarity with other software can be automated.

Manuscript received: 26 February 2010
Communicating Editor: Georg Grossmann

Secondly, there are new requirements with respect to software composition. In the past, composition of software from parts was done by engineers at build-time, before the software was deployed. This was the time to perform verification and validation activities such as integration and acceptance testing. With more and more economic pressure to improve time to market, composition is now often done automatically. In particular, this applies to composition activities related to software maintenance such as the installation of bug fixes. With this being automated, manual pre-deployment verification is often not possible, and new techniques must be introduced to replace it.

There are several new technologies that try to address these challenges in various ways. A term often used in this context when referring to units of functionality provided by software is service. In the context of service-oriented programming, a service is a "contractually defined behaviour that can be implemented and provided by any component for use by any component, based solely on the contract." (Bieber, Architect and Carpenter, 2001). The important part is the emphasis on the separation between the description of services (by contracts), and the actual provision and consumption of services. This is not really new, and has its roots in technology such as forward declaration and header files in C and similar languages, and the use of interface definition languages (IDLs) in remote procedure call (RPC) and related technologies like the common object request broker architecture (CORBA). The focus has now shifted from facilitating the build process and enable transparent distributed computing towards support for runtime composition. Most modern programming languages, in particular Java and C#, have direct support for specifying services by defining them as interfaces. There are several very successful frameworks that use interfaces for runtime composition, such as dependency injection (Fowler, 2004) frameworks like Spring (The Spring Framework, 2010) and Google Guice (Google-Guice, 2010).

However, the focus on interfaces has problems (Beugnard, Jeźequel, Plouzeau and Watkins, 1999). While it supports some form of verification of runtime composites through type checking, other contracts regarding the semantics and quality of service properties of services cannot be expressed. There are several proposals that tackle parts of the problem, such as the use of design by contract (Meyer, 1992) to define and enforce behaviour, but it remains unclear how this can be integrated with dynamic composition. The diversity in ecosystems creates new challenges for service descriptions. For instance, organisations may have constraints with respect to suppliers or licenses of components to be used. A good example are restrictions with respect to the use of licenses that are seen as viral, such as the General Public License (GPL). Quality of service aspects and trust to providers of components are other increasingly important aspects. This suggests that the current practise of describing services as interfaces is not appropriate to address these requirements, and much more expressive means are needed to describe components and their services.

The main contribution of this paper is to propose a simple yet expressive component model centred around the concept of resources and the description of these resources using the standard stack of semantic web languages. This is based on our previous work on contracts for dynamic component systems (Dietrich and Jenson, 2009). We show how existing component models such as OSGi and Eclipse can be significantly simplified and improved by using the proposed model.

The rest of this paper is organised as follows. In Section 2, we introduce a simple component model to represent components and contracts governing the collaboration between these components. In Section 3 we discuss verification based on this model, and show the use of dynamic contract vocabularies. We then discuss in Section 4 metamodelling issues of our approach. In particular, we discuss the differences between our ontology-based approach and the MOF meta model architecture. In Section 5 we present a case study demonstrating how existing component

collaborations in OSGi and Eclipse can be expressed in our model. A discussion of related work and open questions concludes our contribution.

COMPONENTS AND RESOURCES

According to a widely used definition, a software component is "a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition" (Szyperski, 2002). As briefly discussed in the introduction, the interface specification is often done using interfaces of a programming language or a dedicated interface definition language. In an open environment with component repositories without strict quality control, additional information is needed to describe components, including behavioural specification, quality of service and licensing attributes. This is important to safeguard composition, and to build assemblies with context-specific characteristics, such as systems running on embedded hardware with resource restrictions, or mission critical systems requiring high levels of reliability. What is more, describing interfaces using programming language or IDL interfaces is often simply not appropriate. For instance, sometimes a component has to provide structured data. It is much easier to describe this data using a document type definition (DTD) or XML Schema than using an interface of a service that provides this kind of data.

What is therefore needed is a description of component interfaces that does not rely solely on interfaces defined in a programming language type system. The following example taken from Dietrich and Jenson (2009) illustrates the several aspects component interfaces have. Consider a clock component that displays the time. This component uses the services of another date formatter component to display the current time. This component has the responsibility to transform timestamp objects into strings, taking into account local settings and user preferences. The first approach is to describe this with a (Java) interface *DateFormatter* (listing 1).

```
1. interface DateFormatter {
2.    public String format(java.util.Date timestamp);
3. }
```

Listing 1: The DateFormatter interface

This interface merely states that *DateFormatter* has a method accepting timestamps and returning strings. This is not enough to fully express the requirements of the clock component. In particular, the clock component has the following additional requirements:

1. Semantics: the format string should at least print the current hour and minute.
2. Quality of service: to run on mobile devices, the component should not spawn new threads, should not use more than 10ms to compute the string representation of a timestamp, and should not use more than 50k of memory space.
3. Licensing: the formatter component must not be licensed under the GPL.

In Dietrich and Jenson (2009) we have shown how many of these requirements can be expressed using resources. This is trivial for the interface part, if the (Java) interface is considered as a resource. The semantic conditions can be expressed with test suites or constraint sets (such as OCL). The same is true for the quality of service conditions. Finally, the license of the component is a resource that can be directly referenced (as URL) in the component meta data. Requirements with respect to licensing can be expressed in various ways, including resources representing white lists

resource type representing requirement	matching resource type representing capability	relationship	category
Java interface	Java class	implements	syntax
IDL interface	Java class	implements	syntax
XML Schema	XML document	instantiates	syntax
DTD	XML document	instantiates	syntax
JUnit test suite	Java class	passes	semantics, QoS
OCL constraint set	Java class	check succeeds	semantics
White list of licenses	license	contains	licensing
Black list of licenses	license	does not contain	licensing

Table 1: Resources representing requirements and capabilities

of permitted licenses or resources representing black lists of prohibited licenses. Table 1 shows some examples of resources representing requirements and resources representing matching capabilities.

In general, the contractual relationship between components established through their requirements and capabilities can be easily specified using a very simple ontology. This ontology is depicted in Figure 1. The graphical notation used is a modified version of the notation used in Manola and Miller (2004). The UML type stereotype annotations (labels surrounded by guillemets) represent the type of the respective resources. They stand for a *rdf:type* relationship to the resource representing the respective type. The name spaces *rdf*, *rdfs* and *owl* are defined as in Dean and Schreiber (2004), Appendix B. Furthermore, we define *comp* as `http://www.semanticweb.`

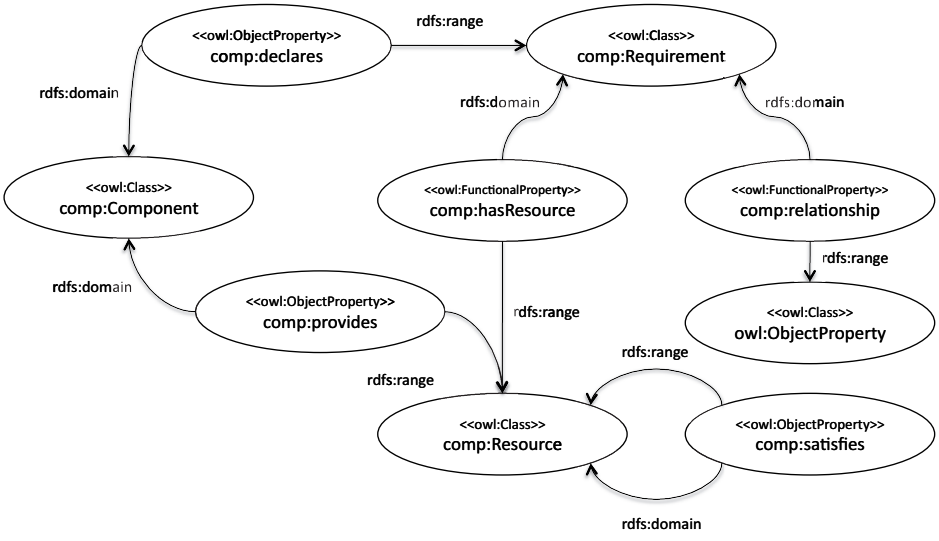

Figure 1: Core Model

org/ontologies/component-core.owl, and *java* as **http://www.semanticweb.org/ontologies/component-java.owl**.

The core model defines only three classes representing components, requirements and resources. Components are associates with resources representing capabilities and requirements, and the *satisfies* relationship expressing that a capability satisfies a requirement. The *Resource* class and the *satisfies* object property are abstract in the sense that they can only be instantiated through subtypes and subproperties, respectively.

This cannot be expressed in OWL directly. We therefore assert the following additional axioms:

Axiom 1 (Abstractness of comp:Resource)
$\forall$r : rdf:type(r,comp:Resource)$\Rightarrow$ $\exists$ c: rdf:type(r,c) $\wedge$rdfs:subClassOf(c,comp:Resource)

Axiom 2 (Abstractness of comp:satisfies)
$\forall r_1, r_2$: comp:satisfies (r_1, r_2) $\Rightarrow$ $\exists$p : rdf:type(p,owl:ObjectProperty)$\wedge$p(r_1, r_2)$\wedge$
rdfs:subPropertyOf(p,comp:satisfies)

To express requirements, associating components with resources is not enough. Some contextual knowledge is needed to describe the role of this resource. This is achieved by replacing the resource by a composite resource representing requirements. This resource also references the object property used to associate capabilities with resources representing requirements. This means that we require the following range restriction for *comp:relationship*:

Axiom 3
$\forall$req, prop :comp:relationship(req, prop) $\Rightarrow$ rdfs:subPropertyOf(prop,comp:satisfies)

We can now formally define a *component configuration* as follows:

Definition 1. A component configuration is a set of components defined using assertions that use the types and predicates defined in the core model including subtypes of comp:Resource and sub properties of *comp:satisfies*, such that the axioms 1-3 are satisfied.

To apply this model to a simple component collaboration consisting of a Java interface and a Java class, the model has to be extended by the respective resource types and the relationship between them. We refer to such an extension as a *vocabulary*. A simple Java vocabulary is shown in Figure 2. *InstantiableClass* represents a class that can be dynamically instantiated. This means

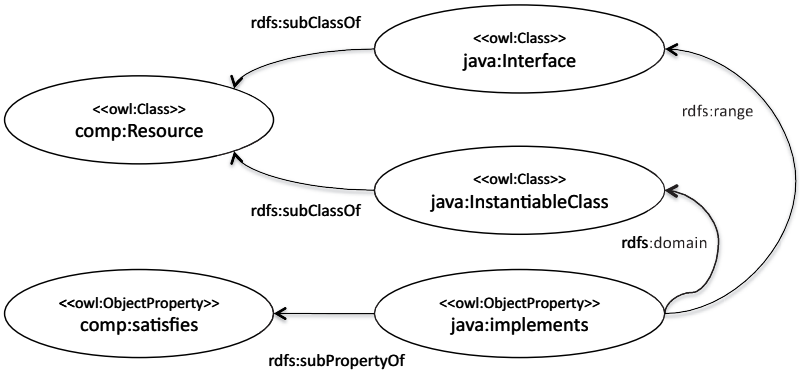

Figure 2: Java vocabulary

that this is a non-abstract class with a public constructor with an empty parameter list. It is very common to make this assumption as client applications only know the interface and have no further knowledge about a particular implementation class. With this additional assumption, classes can be instantiated dynamically through reflection as shown in listing 2. Note that for this to succeed the component container must make sure that the class is accessible by a class loader.

```
1. String     className = ... ;
2. Class serviceProvider = Class.forName(className);
3. ServiceInterface instance =
   (ServiceInterface)serviceProvider.newInstance () ;
```

Listing 2: Dynamic instantiation in Java

Figure 3 shows the use of the respective vocabulary. Two components *clock* and *daterenderer* collaborate through the interface *DateFormatter*. The interface is associated with *clock*. The respective Java byte code file is part of *clock*, and the interface is made available by *clock*. The second component *daterenderer* provides a matching class (*SimpleDateFormatter*). The model also contains a *java:implements* object property that describes the relationship. However, the semantics of this relationship is not defined in the model itself. Semantics refers here to a function that can be used to find out whether for a given pair (*class,interface*)(*class,interface*) ∈ *java:implements* holds. The component container responsible for composition must have knowledge about the semantics of these properties. In this particular case, the container would use the Java Virtual Machine (JVM) to verify that the (compiled) class implements the respective interface. Note that the requirement references the (resource representing the) object property *java:implements*. This is, the requirement implies that the component container responsible for composition has to find another component × that provides a resource *class* such that (*class*, *DateFormatter*) ∈ *java:implements* holds. Then the requirement is matched by a capability and therefore satisfied. In general, this can be expressed for all requirements with the following axiom:

Axiom 4 (Satisfiability)
∀comp, req, res, rel : comp:declares(comp,req) ∧ comp:relationship(req,rel)
∧ comp:hasResource(req,res) ⇒ ∃sup, cap : comp:provides(sup,cap) ∧ rel(cap,res)

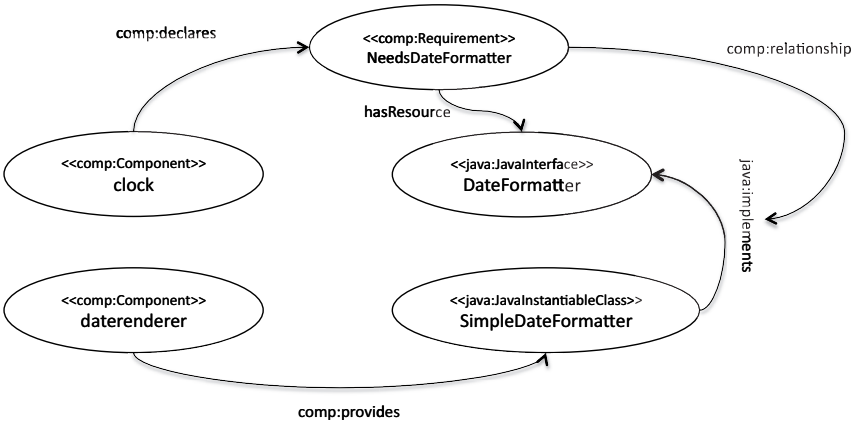

Figure 3: Java vocabulary usage example

Intuitively, axiom 4 states that for each requirement a supplier component (sup) must exist that supplies a capability resource (cap) such that the *rel* condition associates the capability with the requirement resource. Axiom 4 (and axiom 2 likewise) contains quantification about relationships. However, for a given set of components there is only a typically very small fixed set of sub properties of *comp:satisfies*. Hence, quantification could be easily resolved by replacing this rule by a set of separate rules, one for each property.

Using axiom 4 we can precisely define when to consider a component configuration as resolved as follows:

Definition 2. A component configuration is resolved if axiom 4 holds (all requirements are satisfied).

VERIFIERS

The question arises how vocabularies can be integrated into the component model itself. Each vocabulary contributes additional resource types and relationships between those to the core vocabulary defined in Figure 1. The simplest approach would be to restrict the vocabularies to a fixed, predefined set, and to require that each component container responsible for composition has built-in support for these vocabularies. There are however use cases that cannot be addressed in this manner. To take full advantage of ecosystems and the community resources associated with it, systems should follow the invitation rule (Gamma and Beck, 2003). This term has been introduced by the Eclipse community, and states that authors of components should invite others to make contributions by providing extension points. By restricting the expressiveness of how such contributions can be made to a fixed set of predefined techniques, systems cannot take full advantage of the resources of the ecosystem. For instance, a company offering business reporting tools would want customers to contribute custom reporting templates using a product-specific template language. A special template parser would be used to verify the relationship between templates and the template language expressed as a (formal) grammar resource such as an EBNF definition or an ANTLR grammar definition.

In Dietrich and Jenson (2009) we have proposed a solution to this problem for one particular component model (OSGi-Eclipse) by defining a dedicated vocabulary contribution extension point. The Treaty project (Dietrich and Wilke, 2010) contains an implementation demonstrating the benefits of this approach. We generalise this solution by extending the core model proposed in the last section.

We propose to model the vocabulary dependencies as implicit requirements. For this purpose, we extend the core model (Figure 1) as follows:

1. A new class of resources representing verifiers (*comp:Verifier*) is introduced.
2. The *comp:satisfies* property is now also considered to be a resource.
3. A object property *comp:verifies* is introduced that associates verifiers with sub properties of *comp:satisfies*. The respective range restriction is defined in axiom 5.

Figure 4 depicts the extended model. New elements are highlighted with bold face and bold lines, respectively. Axiom 5 formalises the range restriction for the *comp:verifies* property and completes the model.

Axiom 5

$\forall$prop : $\exists$ver : comp:verifies(ver,prop) $\Rightarrow$ rdfs:subPropertyOf(prop,comp:satisfies)

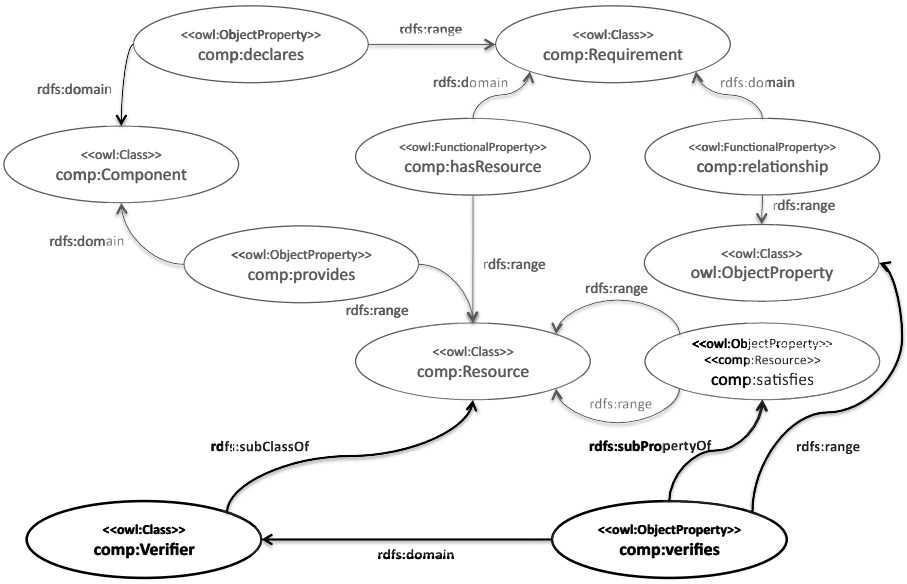

Figure 4: Extended Core Model

The extended core model can now be used to augment component configurations. The purpose of this is to add new requirements stating that verifiers must be available for all conditions used in requirements.

Definition 3. Given a component configuration, the augmentation of the configuration is defined as follows. For each component/requirement combination consisting of a component *comp*, a requirement *req*, a resource *res* and a property *rel* that is different from *comp:verifies*, and defined by the following set of assertions:

1. comp:declares(comp,req)
2. comp:relationship(req,rel)
3. comp:hasResource(req,res)

we add a new requirement *ver* defined by the following set of assertions:

1. comp:declares(comp,ver)
2. comp:relationship(ver,comp:verifies)
3. comp:hasResource(ver,rel)

A model is called *augmented* if it is closed with respect to this rule: applying this rule does not add additional assertions to the model.

The condition that augmentation is only applied for properties that are different from *comp:verifies* makes sure that augmentation is not recursive and therefore terminates. The configuration stays "flat". However, this implies that the mechanism to verify verification must be built into the system. While the verification of other conditions can be delegated to verifiers, a dedicated system verifier is needed for this purpose.

Figure 5 shows augmentation for the Java example used earlier. The requirement for a Java class implementing a Java interface creates the implicit requirement that a tool must be available to check

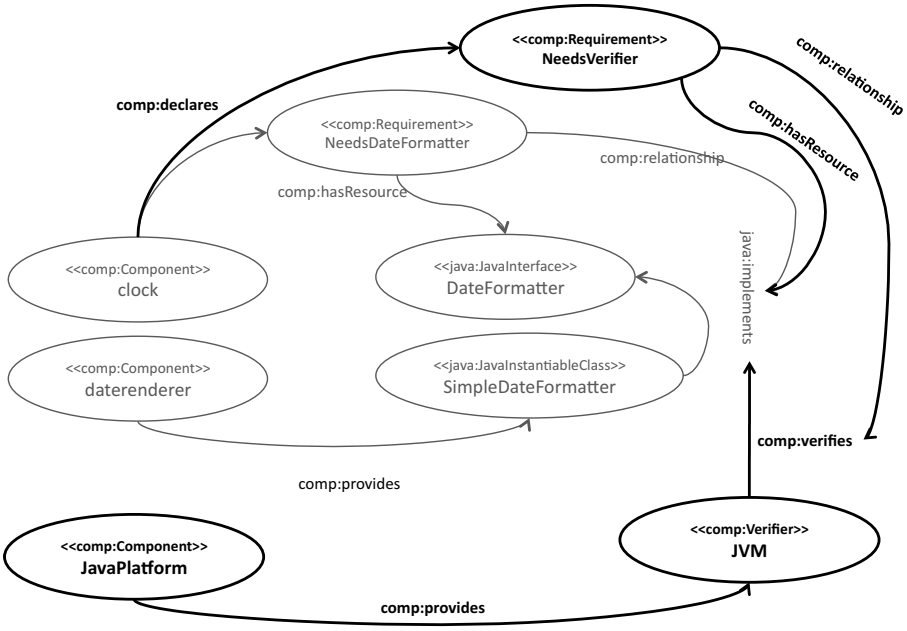

Figure 5: Verifier Dependencies

this relationship. In this example, we assume that there is a JavaPlatform component providing a JVM that can do this.

In the model developed here, the semantics of the component collaborations is mainly defined through the verifiers. The verifiers provide the semantics for the *comp:satisfies* sub properties. The semantics is usually intentional (Carnap, 1947). That means that verifiers will provide a function to compute whether for two given resources r_1, r_2 and a sub property p of *comp:satisfies*, $(r_1, r_2) \in p$ is the case or not[1].

METAMODELLING ISSUES

The question arises whether the use of OWL has advantages over the use of the traditional MOF/UML stack of languages to model component collaborations. Both MOF and OWL address a similar problem: the formal representation of the concepts in a domain in order to describe its semantics. In particular, MOF 2.0 tries to address many of the issues that have been leading to the development of ontology languages. For instance, the MOF specification states that "the Meta Object Facility (MOF), provides a metadata management framework, and a set of metadata services to enable the development and interoperability of model and metadata driven systems" (Meta Object Facility (MOF) Core Specification, version 2.0, 2006). MOF 2.0 has much better support for computational processing of models than previous MOF/UML standards which focused on building models for humans: less precise but more expressive and easy to visualise. It has been noted that both worlds have converged and that there is a significant overlap (Kiko and Atkinson, 2004). It has also been demonstrated that the semantic web technology stack can be fitted into the MOF framework by considering it as model space (Djuric, Gasevic and Devedzic, 2005).

1 As opposed to an extensional semantics where the set of all resource pairs instantiating the predicate is listed.

To check the suitability of a MOF-based model, consider a system that uses two vocabularies: XML and Java. That is, the system has resources including XML files, DTDs, Java objects and interfaces. In terms of MOF, these resources are M0 layer resources. They instantiate M1 types such as custom Java interfaces like *DateFormatter* and classes such as *SimpleDateFormatter*. On the other hand, these artefacts are instances of the entities defined in M2 such as Java classes, Java interfaces, XML documents, DTDs etc. Relationships between these entities are also defined in M2, such as the *implements* relationship between Java classes and instances, and the *instantiates* relationship between XML documents and DTDs.

The component framework proposed here does not fit easily into this layer model. The model layer comprising components and their associated resources would be M1: components collaborate through resources that instantiate types defined in M2. In the case of Java, the resources are indeed Java classes and interfaces, and not instances of these classes. The instantiation of the respective classes is the responsibility of the consumer; the contract must just make sure that the respective class is instantiable. M2 would contain a merged meta-model comprising types from all vocabularies being used in a system. Note that this merging operation must occur at runtime: new components may require support for new types of contracts and therefore new resource types and relationships (defined on M2). On the other hand, components (defined in M1) also reference the relationships linking resources as part of the requirements (defined in M2). This feature reflects the system dynamics of modern component systems: the use of a fixed, built-in set of relationships is not expressive enough.

Using an ontology language provides a solution to this problem: relationships (object properties) can be referenced as resources identified by a unique identifier. This can be seen as a reflection facility similar to features prevalent in modern mainstream programming languages such as Smalltalk. Furthermore, ontology languages provide better runtime support than modeling languages: ontologies can easily be merged at runtime, and queried for consistency and inferred knowledge. The reporting example mentioned above provides an example where this is useful: reporting templates may have to comply to a grammar defined by a template language but restricted by a predefined set of permitted variables - the variables the host application can instantiate. The type of such a template (lets call it *SpecialTemplate*) would be a subtype of the more general template (*GenTemplate*) that does not have the variable restrictions. During verification (at runtime!), the subclass relationship between those types can be used: use the template parser first to find out whether a file is a *GenTemplate*, and if this fails, infer that it cannot be an instance of *SpecialTemplate*.

Not using a layered metamodelling architecture bears risks: the purpose of layers is to provide stratified abstraction. This goes back to Russell's theory of types (Russell, 1937), and is important to avoid logical paradoxes. Additional safeguard mechanisms are needed if strict stratification is not required. The condition that augmentation is only applied to properties that are different from *comp:verifies* is such a condition: if removed, an infinite chain of verifiers would be needed to verify an otherwise finite system.

Case Study: Component Collaborations in the OSGi and Eclipse Component Models

OSGi (Alliance, 2010) is a very successful dynamic component model that is now used in some of the largest and most complex enterprise applications such as IBM WebSphere, Oracle WebLogic (Humble, 2008) and the Eclipse development environment and numerous commercial and open source tools derived from it. In OSGi, components are called *bundles*. Bundles have metadata

associated with them defined in key-value text files, the so-called bundle manifests. The most important part of the manifest is the declaration of capabilities and requirements, both expressed using Java name spaces (packages). Both capabilities and requirements can be annotated with version constraints, and the OSGi container responsible for composition checks these constraints when composition takes place ("wiring"). This solves a particular composition problem known as DLL hell – the availability of multiple, sometimes incompatible versions of the same library leading to runtime errors. Using versions means that additional composition constrains can be expressed and checked as long as the application programmers maps them correctly to versioning constraints. The focus on versioning constraints means that a lot of the responsibility to express contracts is delegated to the application programmer who has to squeeze several types of constraints (quality of service, behaviour, licensing etc) into version dependencies.

```
1. Manifest-Version : 1.0
2. Bundle-SymbolicName : com.acme.bundle1
3. Bundle-Version : 1.0.0
4. Export-Package: com.acme.package1;version=1.3
5. Import-Package: com.acme.package2;version="[1.4.2, 2.0.0)"
6. Require-Bundle: com.acme.bundle2;bundle-version="[1.0.0,2.0.0)"
```

Listing 3: OSGi manifest example

Listing 3 shows an (incomplete) OSGi manifest. The manifest defines a unique name and a version for the bundle, and states that the bundle offers one package (*com.acme.package1*, version 1.3), requires a package (*com.acme.package2*, any version between 1.4.2 inclusive and 2.0.0 exclusive) and another bundle (*com.acme.bundle2*, any version between 1.0.0 inclusive and 2.0.0 exclusive).

This means that bundles offer two kinds of capabilities: a resource of the type *osgi:BundleVersion* that is identified by a unique combination of bundle id and version, and resources of the type *osgi:PackageVersion* identified by unique combinations of package (namespace) name and version. As for the requirements, there are two types of requirements representing bundle and package version ranges. Version ranges are defined by intervals in the totally ordered space of versions consisting of major, minor and micro version numbers. Intervals can be open or closed on either end. The semantics of the satisfies properties *osgi:bundleMatches* and *osgi:packageMatches* is defined as follows: a versioned bundle (package) matches a bundle (package) version range if the names are identical and the version is within the interval defined by the version range. This semantics is enforced by OSGi container. This means that the OSGi container itself provides the verifier.

Resolving these dependencies is a non-trivial task. This is performed in client applications with configurations consisting of hundreds or even thousands of bundles. The same dependency resolution mechanisms are also used to query bundle repositories such as the SpringSource Enterprise Bundle Repository(The SpringSource Enterprise Bundle Repository, 2010) that contains up to tens of thousands of bundles. OSGi containers and repositories often use SAT solvers for dependency resolution.

Figure 6 shows a OSGi vocabulary modeling these collaborations. It uses the same scenario as in listing 3, plus an additional bundle *com.acme.bundle3* that provides the package capability required by bundle *com.acme.bundle1*. The direct bundle dependencies are static. OSGi favours the dynamic dependencies where the consuming component (bundle 1 in the example) does not have

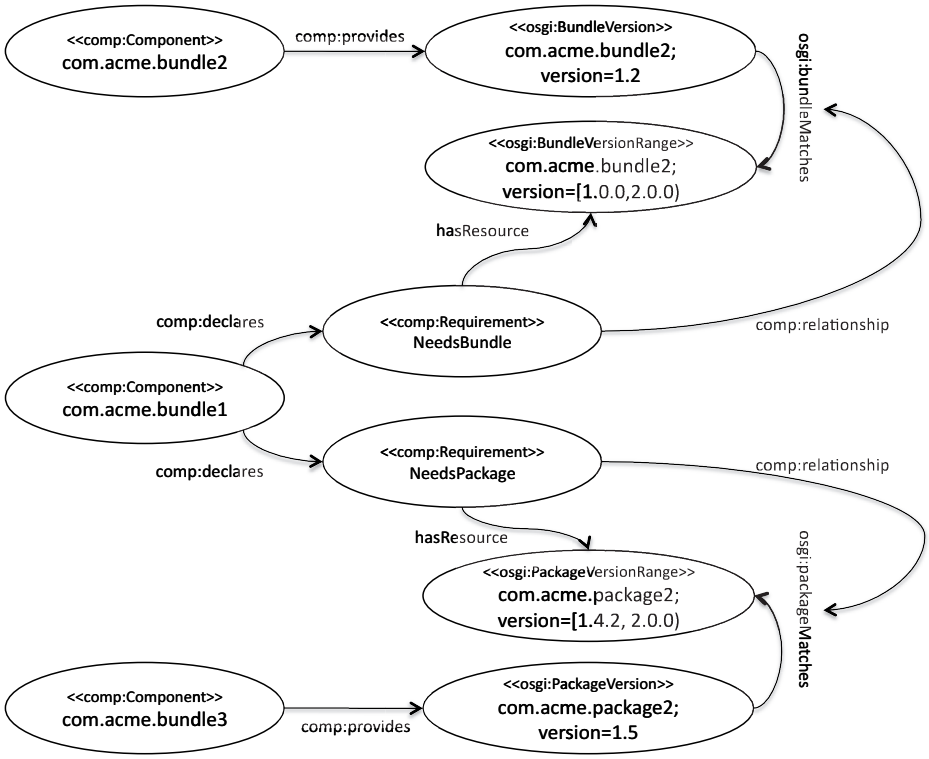

Figure 6: OSGi requirements and capabilities

direct knowledge about the component providing the capability (bundle 3).

Several extensions of OSGi provide support for service-oriented programming. Services are defined by interfaces, and bundles have additional meta-data to define required and provided services. Examples are Declarative Services (Alliance, 2010), Spring Dynamic Modules (Spring Dynamic Modules for OSGi™ Service Platforms, 2010) and Eclipse (The Eclipse project, 2010). Eclipse is particularly interesting as it generalises the concept of services and allows the declaration and provision of other resources as well. This is driven by use cases such as a modular help system. Here, bundles (called plugins in Eclipse) do not have to provide Java classes to make contributions to the help system. Instead, they provide a table of content XML file that has links to the actual help pages.

The collaboration of Eclipse components is organised in two separate layers. First, Eclipse components are OSGi bundles, provide OSGi metadata and are composed by Equinox, the OSGi implementation used by Eclipse. On top of this, plugins advertise requirements through extension points and offer capabilities through extensions for these extension points. This part of the meta data is defined in a separate file plugin.xml. The composition based on matching extension points is done by the extension registry. This dual approach has historical reasons – earlier versions of Eclipse were not based on OSGi.

Listing 4 shows part of the plugin metadata for the plugin *org.tigris.subversion.subclipse.doc 1.3.0*. This plugin provides the documentation for the Subclipse subversion client for Eclipse. The

plugin does not need to provide (Java) code, instead, it provides contributions to the help system – a set of html pages indexed by a table of content (TOC) xml file. The structure of the XML used to define the metadata is defined as part of the extension point *org.eclipse.help.toc* that is being extended. This definition is semi-formal. In particular, the document type definition (DTD) that constraints the structure of the XML to be used in the toc files contributed by extensions is only defined as text embedded in the documentation of the extension point[2].

By not making this requirement explicit, the responsibility to check that plugins provide correct extensions is with the consuming plugin. In particular, the code to parse the toc contributions must enforce the rules defined in the DTD. It turns out that this is not the case. In particular, Eclipse also accepts additional markup used in toc file and does not strictly enforce the contract defined in the extension point documentation. Examples where violations occur are the *org.eclipse.platform.doc.user* plugin (an additional *enablement* element is used – 3 violations), and the *org.eclipse.jdt.doc.user* extension point (an additional *extradir* attribute is used – 2 violations). This is in direct violation of one of Eclipse community house rules (Gamma and Beck, 2003), the fair play rule. This rule states that insider knowledge must not be used to implement extensions, everybody must rely only on the external interfaces of plugins. This is directly related to creating a level playing field for all parties contributing to the ecosystem. However, insider knowledge has been used here in plugins by people knowing that the syntax of the toc files is not strictly enforced, and that the application will use (interpret) additional markup such as the *enablement* element and *extradir* attribute.

By using the approach proposed in this paper this can be easily avoided: the requirement is a resource of the type DTD, the *satisfies* relationship is the implementation of a DTD by an XML document, the capability is a toc XML document and the verifier is a validating XML parser.

```
1. <?xml version="1.0" encoding="UTF?8"?>
2. <?eclipse version="3.0"?>
3. <plugin>
4. <extension point="org.eclipse.help.toc"> file="toc.xml" />
5. <tocfile="tocgettingstarted .xml"    />
6. <toc file="tocreference.xml" />
7. <tocfile="dailywork.xml" />
8. <tocfile="tocplatformGettingstarted.xml" />
9. </extension>
10.</plugin>
```

Listing 4: An Eclipse plugin manifest

FUTURE AND RELATED WORK, CONCLUSION

We have presented a simple ontology-based model for component collaborations. The proposed ontology is open in the sense that it can be and actually has to be extended, and lightweight in the sense that it defines only a very small number of types and relationships. To achieve this, many concepts that are used inside the ontology, in particular the satisfies relationships and the respective domain and range types, are actually defined outside the ontology. The ontology contains merely views on those artefacts but does not define their semantics. We believe that this approach is

2. http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.platform.doc. isv/reference/extension-points/org_eclipse_help_toc.html

beneficial in the long term as the semantics of some of the relationships is very complex and difficult to model directly in the ontology. An example are the rules used by the Java compiler or by the Java virtual machine to decide whether a class implements an interface. This involves the check of method signatures including exception types, consistence of type parameters (for the compiler only) including checks for co- and contravariance and so on. Our approach is similar to the way RDF is often used. It turns out that representing large amounts of data in RDF directly is very difficult to manage, and that it is more realistic to expose existing data stored in traditional databases as RDF through scripts.

Our approach has similarities with efforts to add semantics to web services (Burstein, Hobbs, Lassila, Mcdermott, Mcilraith, Narayanan, Paolucci, Parsia, Payne, Sirin, Srinivasan and Sycara, 2004). However, our model can be simpler. In particular, we do not try to model service grounding as this is part of the component model used by the component container responsible for composition. Because we assume that composition is automated, we do not attempt to model human-readable component descriptions (profile). Many component models however have metadata entries providing these capabilities.

In Oberle, Lamparter, Grimm, Vrandecíc, Staab and Gangemi (2006) the authors propose a comprehensive ontology to describe large systems composed from modules. In this ontology, component interfaces are described using concepts extracted from object-oriented programming such as methods and their associated return, parameter and exception types. Our ontology is less fine-grained. In particular, type reasoning about methods is not part of the ontology but delegated to verifiers.

In Shahri, Hendler and Porter (2007) the authors use ontologies for software configuration management. The scope of this work is much narrower than our approach and focuses on the resolution of version constraints. The case study on OSGi shows how this can be integrated into the framework proposed here.

In Pahl (2007), the author presents an ontology that is formalised interface matching based on design-by-contract. Our approach is more general as it covers other forms of composition that are not based on matching interface descriptions.

A limitation of the current approach is that we assume that components always have a fixed set of mandatory requirements. This could be extended by allowing optional requirements and alternatives. An example how this can be used in a concrete component models (Eclipse) is given in Dietrich and Jenson (2009).

REFERENCES

ALLIANCE, T.O. (2010): OSGi service platform core specification. Release 4, Version 4.2 June 2009. Accessed 21 July 2010. URL: http://www.osgi.org/Download/File?url=/download/r4v42/r4.core.pdf

BEUGNARD, A., JEŹEQUEL, J-M., PLOUZEAU, N. and WATKINS, D. (1999): Making components contract aware, *Computer* 32(7): 38–45.

BIEBER, G., ARCHITECT, L. and CARPENTER, J. (2001): Introduction to service-oriented programming, in *Openwings*, URL = http://www.openwings.org.

BURSTEIN, M., HOBBS, J., LASSILA, O., McDERMOTT, D., McILRAITH, S., NARAYANAN, S., PAOLUCCI, M., PARSIA, B., PAYNE, T., SIRIN, E., SRINIVASAN, N. and SYCARA, K. (2004): OWL-S: Semantic markup for web services. Accessed 21 July 2010. URL: http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/

CARNAP, R. (1947): Meaning and necessity, University of Chicago Press.

DEAN, M. and SCHREIBER, G. (2004): OWL web ontology language reference, W3C recommendation, W3C.

DIETRICH, J. and JENSON, G. (2009): Components, contracts and vocabularies – making dynamic component assemblies more predictable, *Journal of Object Technology* 8(7): 131–148.

DIETRICH, J. and WILKE, C. (2010): Treaty – A framework for contracts in dynamic component-based systems. Accessed 21 July 2010. URL: http://code.google.com/p/treaty/

DJURIC, D., GASEVIC, D. and DEVEDZIC, V. (2005): Adventures in modeling spaces: Close encounters of the semantic web and mda kinds, in KENDALL, E.F., OBERLE, D., PAN, J.Z. and TETLOW, P. eds, *Workshop on Semantic Web Enabled Software Engineering (SWESE 2005)*, Galway, Ireland.
FOWLER, M. (2004): Inversion of control containers and the dependency injection pattern. Accessed 21 July 2010. URL: http://martinfowler.com/articles/injection.html
GAMMA, E. and BECK, K. (2003): Contributing to Eclipse. Principles, patterns and plugins, Addison-Wesley.
GOOGLE-GUICE (2010): Accessed 21 July 2010. URL: http://code.google.com/p/google-guice/
HUMBLE, C. (2008): IBM, BEA and JBoss adopting OSGi. Accessed 21 July 2010.URL: http://www.infoq.com/news/2008/02/osgi_jee
KIKO, K. and ATKINSON, C. (2004): Integrating enterprise information representation languages, in *International Workshop on Vocabularies, Ontologies and Rules for The Enterprise (VORTE)*. URL: http://www.ctit.utwente.nl/library/proceedings/vorte.pdf
MANOLA, F. and MILLER, E., eds (2004): RDF Primer, W3C Recommendation, World Wide Web Consortium. Accessed 21 July 2010. URL: http://www.w3.org/TR/rdf-primer/
META OBJECT FACILITY (MOF) Core Specification, version 2.0 (2006): Accessed 21 July 2010. URL: http://www.omg.org/spec/MOF/2.0/
MEYER, B. (1992): Applying "Design by Contract", *Computer* 25(10): 40–51.
OBERLE, D., LAMPARTER, S., GRIMM, S., VRANDECIC, D., STAAB, S. and GANGEMI, A. (2006): Towards ontologies for formalizing modularization and communication in large software systems, *Appl. Ontol.* 1(2): 163–202.
PAHL, C. (2007): An ontology for software component matching, *Int. J. Softw. Tools Technol. Transf.* 9(2): 169–178.
RUSSELL, B. (1937): Principles of mathematics, 2nd edn, G. Allen &Unwin, London.
SHAHRI, H.H., HENDLER, J.A. and PORTER, A.A. (2007): Software configuration management using ontologies, in *Proceedings of the 3rd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2007)*.
SPRING DYNAMIC MODULES FOR OSGITMSERVICE PLATFORMS (2010): Accessed 21 July 2010. URL: http://www.springsource.org/osgi
SZYPERSKI, C. (2002): Component software: Beyond object-oriented programming, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
THE ECLIPSE PROJECT (2010): Accessed 21 July 2010.URL: http://www.eclipse.org/
THE SPRING FRAMEWORK (2010): Accessed 21 July 2010.URL: http://www.springsource.org
THE SPRINGSOURCE ENTERPRISE BUNDLE REPOSITORY (2010): Accessed 21 July 2010. URL: http://www.springsource.com/repository/app/

BIOGRAPHICAL NOTES

Jens Dietrich is an associate professor at the School of Engineering and Advanced Technology (SEAT) at the Palmerston North Campus of Massey University in New Zealand. He has a Master in Mathematics and a PhD in Computer Science from the University of Leipzig in Germany. After completing his PhD in 1996, he worked as a consultant in Germany, Namibia, Switzerland and the UK, and returned in 2003 to academia. His research interests are in the areas of design pattern formalisation, software componentry, motif detection in graphs and business rule automation.


Jens Dietrich