The Knowledge Engineering Review, Vol. 24:1, 59–75. © 2009, Cambridge University Press
doi:10.1017/S0269888909000137 Printed in the United Kingdom

A semantic web environment for components

HAI H. WANG[1] and JING SUN[2]

[1]*School of Engineering and Applied Science, Aston University, United Kingdom;*
e-mail: h.wang@aston.ac.uk
[2]*Department of Computer Science, The University of Auckland, Auckland, New Zealand;*
e-mail: j.sun@cs.auckland.ac.nz

Abstract

Component-based development (CBD) has become an important emerging topic in the software engineering field. It promises long-sought-after benefits such as increased software reuse, reduced development time to market and, hence, reduced software production cost. Despite the huge potential, the lack of reasoning support and development environment of component modeling and verification may hinder its development. Methods and tools that can support component model analysis are highly appreciated by industry. Such a tool support should be fully automated as well as efficient. At the same time, the reasoning tool should scale up well as it may need to handle hundreds or even thousands of components that a modern software system may have. Furthermore, a distributed environment that can effectively manage and compose components is also desirable. In this paper, we present an approach to the modeling and verification of a newly proposed component model using Semantic Web languages and their reasoning tools. We use the Web Ontology Language and the Semantic Web Rule Language to precisely capture the inter-relationships and constraints among the entities in a component model. Semantic Web reasoning tools are deployed to perform automated analysis support of the component models. Moreover, we also proposed a service-oriented architecture (SOA)-based semantic web environment for CBD. The adoption of Semantic Web services and SOA make our component environment more reusable, scalable, dynamic and adaptive.

1 Introduction

Component-based development (CBD) (Szyperski, 1998) is an important emerging topic in software engineering, which aims to compose systems from pre-built software units and sub-modules. A CBD system is developed as a composite of subparts rather than as a monolithic entity. Components are replaceable units in systems, thus facilitating the development of software systems that are more flexible and less likely to become prematurely obsolete. There are many benefits of a component-based approach to software development, such as increased software reuse, reduced development time to market, and, hence, reduced software production cost. Components and industry standard component models provide a basis for commerce in reusable software.

Recently, a novel component model (Lau *et al.*, 2005) has been proposed, which could be considered as a prominent improvement over the existing CBD. In this model, a special kind of component interaction operator called *exogenous connectors* was proposed for component composition. The exogenous connector provides a composition mechanism different from that in existing component models, in which it concentrates on capturing control interactions, leaving components to encapsulate only computation. This makes components truly independent and therefore more reusable in different software architectures. An exogenous connector also makes hierarchical system design possible, owing to its own strictly hierarchical nature. It makes system construction easier by enabling automated control flow generation from its architecture. Systems

based on this component model are easier to manage because they are modular and maintainable. Applications that used the component model have proved to be more efficient and cost effective (Lau *et al*., 2005; Lau *et al*., 2006b; Lau & Wang, 2007).

Despite these potential benefits, one critical aspect of the exogenous-connector component modeling ought to be adequately addressed, that is, the need for tool support that can effectively assist developers to manage components and develop the composition. It is highly desirable to have methods and tools that can support automated analysis over the component model. In general, such methods and tools should provide us with a means of verifying the correctness of a component composition, as the design of a component-based software may be inconsistent. Once we have chosen a combination of components for a specific software product, such tools should be able to check the correctness of the combination based on the constraints defined in the component model. Industrial experiences show that in large-scale software development, the number of components could be thousands, and the composition could be very complicated. As a result, a substantial amount of effort is spent on correcting these human errors. In detail, such a tool support should bear the following number of requirements:

Automated inconsistency detection: Different compositions may be invalid with respect to the component model. To prevent invalid compositions from being combined to incompatible components, it is important that those inconsistencies be detected automatically. It should allow the domain experts to focus only on the system to be built, rather than the usability of the tool. Furthermore, the automation should also enable computer agents to compose software in runtime based on users' demands.
Reasoning efficiency: As a component-based application may evolve constantly, especially for the dynamic re-configured component systems, it requires the component-reasoning tool to be able to conclude the validity of configurations in a short period of time.
Scalability: Modern component-based software systems could be very complex in terms of their size. The manual checking of such compositions is highly painstaking and error-prone. Hence, the component-reasoning tool should scale-up well to handle large and complex models.
Expressivity: The reasoning system should provide a good means of representing and efficiently reasoning over a wide variety of composition relationships.

The Semantic Web has emerged as the next generation of the Web. Ontology languages, such as OWL (McGuinness & van Harmelen, 2003), play a key role in realizing the full potential of the Semantic Web, as they prescribe meaningfully how data are defined and inter-related. According to W3C, '*an ontology defines the terms used to describe and represent an area of knowledge… Ontologies include computer-usable definitions of basic concepts in the domain and the relationships among them… They encode knowledge in a domain and also knowledge that spans domains. In this way, they make that knowledge reusable*'. One of the advantages of logic-based ontology languages, such as OWL and its rule extension Semantic Web Rule Language (SWRL) (Horrocks *et al*., 2003), is that reasoners can be used to compute subsumption relationships between classes and identify unsatisfiable (inconsistent) classes. With the maturation of tableaux algorithm-based DL reasoners, such as **RACER** (Haarslev & Möller, 2001), **FaCT++** (Horrocks, n.d.) and **PELLET** (Evren Sirin, 2004), it is possible to perform efficient reasoning on large-scale ontology formulated in expressive description logics.

In this paper, we explore the synergy of component modeling and the Semantic Web. Given the rich expressiveness of OWL and SWRL, and their efficient and automated reasoning support, the ontology can be adopted to reason and verify component composition effectively. We define a knowledge model (ontology) for the exogenous-connector component model and use OWL+SWRL reasoning engine such as, emphRacer or *Jess* (Laboratories, 1991), to perform automated analysis over the defined component model. The analysis helps us detect possible inconsistencies in component composition and identify the candidate component to be composed. We illustrate our approach using an example of the bank component model, which is a commonly used problem proposed in Lau *et al*. (2005) for evaluating CBD. Furthermore, by combining the ontology we developed for the component model and the latest Semantic Web Service (SWS)

techniques, we also propose a service-oriented architecture (SOA)-based Web environment for CBD. The adoption of SWSs and SOA makes this component environment more reusable, scalable, dynamic, and adaptive.

The remainder of the paper is organized as follows. Section 2 gives a brief overview of the exogenous-connector component model and Semantic OWLs and tools. Sections 3 and 4 describe the ontology model of the component and demonstrate how the Semantic Web reasoning engines can be used to perform automated analysis over the component model. Section 5 concludes the paper and describes future works.

2 Overview

2.1 Component model and exogenous connectors

There are many definitions regarding components in the software engineering community, some of which are summarized below:

- 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 composition by third parties (Szyperski, 1998).
- A component is a software element (modular unit) that can be used by other software elements and its clients, possesses an official usage description, which is sufficient for a client author to use and is not tied to any fixed set of clients (Meyer, 2003).

In a commonly accepted view of current software component modeling, components are typically objects as in object-oriented languages, and port-connector types are architectural units with method calls and Architecture Description Languages (ADL; Shaw & Garlan, 1996) connectors as composition mechanisms. A detailed survey of current component models can be found in Lau and Wang (2007).

Recently, Lau *et al.* (2005) have proposed a novel component model in which components encapsulate computation and connectors encapsulate control. The most important innovation of this model is that in this model, components are not allowed to invoke methods in other components, and control originates in and flows from connectors, leaving components to encapsulate only computation. All method calls are initiated and coordinated by exogenous connectors, which thus encapsulate control, i.e. they initiate and coordinate control.

There are two main constructs in this model:

Computation units, which performs computation only within itself and cannot invoke computation in other units.

exogenous connectors, which initiate and coordinate all controls (method calls) connecting the components. Exogenous connectors are organized as a hierarchy structure. At the bottom of the hierarchy, an exogenous method, *invocation connector*, is used. It is a unary operator that takes a component, invokes one of its methods, and receives the result of the invocation. To structure the control flow in a set of components or a system, other connectors for sequencing exogenous method calls to different components are defined in the next levels of the type hierarchy. There are n-ary connectors for connecting invocation connectors. The model defines and implements pipe connectors (for sequencing) and selector connectors (for branching). Using the exogenous connection, where all method calls are initiated and coordinated by exogenous connectors, it allows us to separate control from computation in component-based systems, which makes components truly independent and, therefore, more reusable in different architectures.

2.2 The Semantic Web

2.2.1 Ontology languages

The Semantic Web is a vision for a next generation of Web with enhanced functionality, which will require semantic-based representation and processing of Web information. W3C has proposed a

series of technologies that can be applied to achieve this vision. The Semantic Web extends the current Web by giving the web content a well-defined meaning, better enabling computers and people to work in cooperation. XML is aimed at delivering data to systems that can understand and interpret the information. XML is focused on the syntax (defined by the XML schema or DTD) of a document and it essentially provides a mechanism to declare and use simple data structures. However, there is no way for a program to actually understand the knowledge contained in the XML documents. Resource Description Framework (RDF) (Lassila & (eds.), 1999) is a foundation for processing metadata; it provides interoperability between applications that exchange machine-understandable information on the Web. RDF uses XML to exchange descriptions of Web resources and emphasizes facilities to enable automated processing. The RDF descriptions provide a simple ontology system to support the exchange of knowledge and semantic information on the Web. RDF Schema (Brickley & Guha, 2004) provides the basic vocabulary to describe RDF documents. RDF Schema can be used to define properties and types of the web resources.

OWL is the latest standard in ontology languages, which was developed by members of the World Wide Web Consortium (W3C) and the DL community. An OWL ontology consists of classes, properties and individuals. Classes are interpreted as sets of objects that represent the individuals in the domain of discourse. Properties are binary relations that link individuals, and are interpreted as sets of tuples, which are subsets of the cross product of the objects in the domain of discourse. Although OWL includes a relatively rich set of class constructors, the language provided for expressing properties is much weaker. SWRL (Horrocks *et al.*, 2003) intends to overcome the expressive restriction of OWL properties by extending OWL with some form of 'rule language'. SWRL is based on a combination of the OWL DL and OWL Lite sub-languages of OWL with the Unary/Binary Catalog sub-languages of the Rule Markup Language. SWRL introduces a high-level abstract syntax for Horn-like rules in both the OWL DL and OWL Lite sub-languages of OWL. SWRL extends OWL by also allowing rule axioms, i.e., by adding the construct:

$$axiom ::= rule$$

A rule axiom consists of an antecedent and a consequent, each of which consists of a set of atoms that could be class membership ($C(x)$), property membership ($P(x,y)$) or individual in/equalities ($differentFrom(x,y)/sameAs(x,y)$). Informally, a rule means that if the antecedent holds (is 'true'), then the consequent must also hold. A simple example of the rules could be used to express the knowledge that 'if $?x1$ is a child of $?x2$ and $?x2$ is a brother of $?x3$, then $?x3$ is an uncle of $?x1$'. Informally, this rule could be written as

$$hasChild(?x2, ?x1) \wedge hasBrother(?x2, ?x3) \rightarrow hasUncle(?x1, ?x3)$$

Due to the highly expressive needs of the component model, we will rely on the SWRL rules to capture some of the model constraints.

2.2.2 Ontology-reasoning tools

Many ontology-related tools have been built alongside the development of ontology languages. FaCT++ (*Fa*st *C*lassification of *T*erminologies) (Horrocks, n.d.), Pellet (Evren Sirin, 2004), RACER (*R*enamed *A*Box and *C*oncept *E*xpression *R*easoner) (Haarslev & Möller, 2001) and Jess (Laboratories, 1991) are the most widely accepted OWL and SWRL reasoners. They support automated class subsumption and consistency reasoning and some queries on OWL ontologies. In this paper, Jess will be used as an example to show some of the reasoning tasks. Jess (Java Expert System Shell) (Laboratories, 1991) is a rule engine and scripting environment written in the Java language. It was inspired by the production rule language and system, CLIPS, and it sports an API for creating rule-based expert systems in Java. Jess has basically three components: a rule base, a working memory that contains all the facts and an inference engine that matches rules against the facts.

Protégé (Gennari *et al*., 2002) is a system for developing knowledge-based systems. It is an open-source, Java-based Semantic Web ontology editor that provides an extensible architecture, allowing users to create customized applications. In particular, the Protégé-OWL plugin (Knublauch *et al*., 2004) enables editing OWL ontologies and connecting to reasoning engines to perform tasks such as automated consistency checking and ontology classification. The Protégé plugin SWRLJessTab[1] bridges the reasoning power of the Jess inference engine for SWRL ontologies through the plugin architecture (Knublauch *et al*., 2004) of Protéé-OWL. It translates SWRL rules and the OWL ontology into Jess format, making them available to be reasoned about. The reasoning result (Jess assertions representing new facts) can subsequently be asserted back to the ontology.

3 Ontology for component modeling

In this section, we describe how to build a Semantic Web ontology framework for the exogenous-connector component model. Various entities and relations within the component model are modeled using OWL and SWRL language constructs.

Modeling the component model with an Ontology approach has the following advantages: First, it can facilitate component model storing, sharing and distributing, retrieving and assisting cooperative designing. Second, it can provide precise and lightweight semantics of the component model. Finally, we can make use of the semantic web reasoning tools to provide an automated verifying support for the component model design.

For brevity reasons, we omit all OWL namespace definitions in this paper. Moreover, we assume that all classes are mutually disjoint and individuals with different names are not same. This is because OWL assumes that any two names may represent the same entity unless explicitly stating that they are different, and that any two classes may overlap unless they are stated to be disjoint. Such disjointness statements are also omitted in the paper.

3.1 Computation units

In the exogenous-connector component model, components are constructed from *computation units* and *connectors*. A *computation unit* can be a class, module, Oracle package, etc. constrained to perform computations within its boundaries. We model it as

```
Class(ComputationUnit)  Class(Method)
ObjectProperty(hasMethods InverseFunctional
  domain(ComputationUnit)  range(Method))
```

The above defines a typical computation unit in the component model. *ComputationUnit* and *Method* are modeled as OWL classes. For a computation unit, the object property, *hasMethods*, denotes the set of methods that it can invoke. It should not invoke other units' methods external to its boundaries. Therefore, the property, *hasMethods*, is declared to be 'InverseFunctional' (the inverse property of this OWL property is functional).

3.2 Connectors

Connectors are used to construct atomic components from computation units, and compose them to form composite components. Each connector has a *level* in the composition hierarchy, which is denoted by the OWL Datatype property, *hasLevel*, with integer as *range*.

```
Class(Connector)
DatatypeProperty(hasLevel functional range(Integer))
```

[1] Please refer to `http://protege.cim3.net/cgi-bin/wiki.pl?SWRLJessTab` for more details.

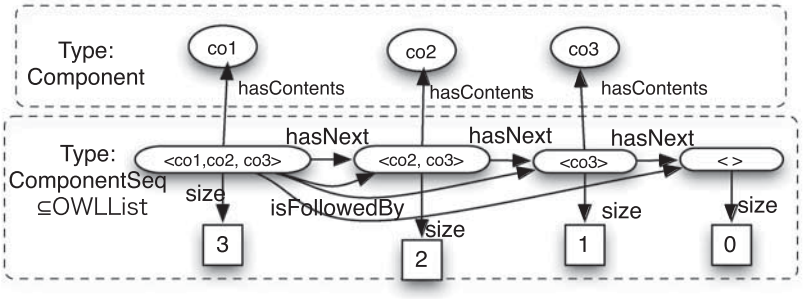

Figure 1 OWL model of component sequence

Invocation connector, as a special kind of connector, is used to create atomic components, and each invocation connector is attached (modeled by *hasCompUnit*) to one *computation unit* for invoking one of its methods. The *level* value for an invocation connector must be 1.

```
ObjectProperty(hasCompUnit functional
  domain(InvocationConnector) range(ComputationUnit))
Class(InvocationConnector partial Connector
  restriction(hasCompUnit cardinality(1))
  restriction(hasLevel hasValueof(1)))
```

Another important kind of connector is the *composition connector*, which is used to compose complicated components from one or more other components. Before we model the *composition connector* in OWL, we first define some necessary entities. Note that the numbers inserted at the beginning of each line code is merely for easier reference and they are not a part of the OWL model. The OWL functional property, *hasCompSeq* (lines 1–3), denotes the sequence of components (denoted by *ComponentSeq*) that are composed by a composition connector. As OWL contains no specific support for ordering, we adopt a well-accepted design pattern for modeling the sequence of components within OWL-DL (Drummond *et al.*, 2006) (*Component* will be modeled later). In this pattern, several OWL properties and classes (e.g. *OWLList*, *isFollowedBy*, *hasContents*, etc.) are predefined for modeling ordered elements. Figure 1 shows an example of a component sequence that follows this modeling pattern, in which each item is held in a cell (*ComponentSeq*, as a subclass of *OWLList*). Each cell has three pointers, one to a head (*hasContents*), one to the tail cells (*hasNext*) and one to an integer denoting the size of the sequence; the end of the list is indicated by a terminator (*EmptyList*) that also serves to represent the empty list. For convenience, we also define an extra property, *allListContents*, to denote all the elements contained within a sequence. The two SWRL rules (lines 5 and 6) are used to define the relationship between *allListContents* and *OWLList*. More information about this modeling pattern can be found in Drummond *et al.* (2006).

```
(1)  ObjectProperty(hasCompSeq functional
(2)    domain(CompositionConnector)
(3)    range(ComponentSeq))
(4)  ObjectProperty(allListContents)
(5)  OWLList(?x) ∧ hasContents(?x, ?y) →
           allListContents(?x, ?y)
(6)  OWLList(?x) ∧ isFollowedBy(?x, ?y) ∧
           hasContents(?y, ?z) → allListContents(?x, ?z)
(7)  Class(ComponentSeq partial OWLList
(8)    restriction(hasContents AllValuesFrom(Component))
(9)    restriction(isFollowedBy
           AllValuesFrom(ComponentSeq)))
```

The *composition connector* below is defined as a subclass of *Connector*, which can only appear at Level 2 or higher in the composition hierarchy. This is modeled by an *AllValuesFrom* restriction (line 2). Note that '*[2 ...]*'[2] denotes the integers that are larger or equal to 2. Furthermore, a composition must involve at least two components (line 4). The minCardinality is 2 because there is always an empty list as the tail of a sequence.

```
(1)  Class(CompositionConnector partial Connector
(2)    restriction(hasLevel AllValuesFrom([2 ...]))
(3)    restriction(hasCompSeq cardinality(1))
(4)    restriction(hasCompSeq
           AllValuesFrom(intersectionOf(
             ComponentSeq
         restriction(isFollowedBy minCardinality(2))))))
```

CompositionConnector also has some additional constraints, such as all-child components of a composition connector (i.e., all elements of the component sequence) have to be distinct. This is modeled by the following SWRL rules:

```
hasCompSeq(?x, ?y) ∧ hasContents(?y, ?c1) ∧
  isFollowedBy(?x, ?y) ∧ hasContents(?z, ?c2)
→ differentFrom(?c1, ?c2)
hasCompSeq(?x, ?y) ∧ isFollowedBy(?y, ?z1) ∧
  isFollowedBy(?y, ?z2) ∧ hasContents(?z1, ?c1) ∧
  hasContents(?z2, ?c2) ∧ differentFrom(?z1, ?z2)
→ differentFrom(?c1, ?c2)
```

Furthermore, any two-child component of a composition connector must not contain the same instances (i.e., computation units or connectors) and the top-most composition connector itself must not be used in any of its child components. The following two rules model these constraints (*hasInst* will be defined later):

```
hasCompSeq(?x, ?y) ∧ allListContents(?y, ?c1) ∧
  allListContents(?y, ?c2) ∧ hasInst(?c1, ?i1) ∧
  hasInst(?c2, ?i2) ∧ differentFrom(?c1, ?c2)
    → differentFrom(?i1, ?i2)
hasCompSeq(?x, ?y) ∧ allListContents(?y, ?c) ∧
  hasInst(?c, ?i) → differentFrom(?x, ?i)
```

Finally, the *level* of a composition connector must match the composition hierarchy, which is modeled by the following two SWRL rules. The first rule shows that a composition connector's *level* must be larger than its child components' level. The second rule denotes that for any composition connecter, there must exist one child component whose level is only 1 less than the composite connector's level.

```
CompositionConnector(?x) ∧ hasLevel(?x, ?l1) ∧
  hasCompSeq(?x, ?y) ∧ allListContents(?y, ?c) ∧
  hasLevel(?c, ?l2) → swrlb:greaterThan(?l1, ?l2)
CompositionConnector(?x) ∧ hasLevel(?x, ?l1)
  ∧ hasCompSeq(?x, ?y) ∧ swrlb:add(?l1, ?l2, 1)
  →restriction(allListContents someValuesFrom(
  intersectionOf(Component
  restriction(hasLevel hasValueof(l2)))(?y)
```

[2] OWL 1.1 allows such kind of user-defined datatypes, see at http://www.webont.org/owl/1.1.

There exist different types of composition connectors, with different implementations of their ways to execution methods. For example, a *numerical selector* is a connector whose execute method takes a number *i* as a parameter and invokes the *i*th child component, and an *open pipe* is a connector which invokes all-child components in sequence, piping the output of one invocation to the next one. *numerical selector*, *open pipe* and other kinds of specific-composition connectors can be extended from *CompositionConnector* as follows:

```
Class(NumSelector partial CompositionConnector)
Class(OpenPipe partial CompositionConnector)
```

3.3 Components

Having defined all the necessary types, we can now provide a definition of components. We model it as an OWL class. Similar to *connectors*, a component also uses *hasLevel* to denote its position in the composition hierarchy. *hasConnector* is used to link a component and its top-level connector and *hasInst* gives all instances (connector and computation unit) included in a component.

```
ObjectProperty(hasConnector functional
  domain(Component) range(Connector))
ObjectProperty(hasInst domain(Component))
Class(Component
  restriction(hasConnector cardinality(1))
  restriction(hasLevel cardinality(1)))
```

There are two different kinds of components: *atomic components* and *composite components*. An atomic component, modeled as a subclass of *Component*, is constructed from a computation unit and from an invocation connector (invoker for short). The invoker exposes an interface for the atomic component. For an atomic component, the instances included in it are the invocation connector and the computation unit, which is captured by the last two SWRL rules.

```
Class(AtomComp partial Component
  restriction(hasConnector
    allValuesFrom(InvocationConnector))
  restriction(hasLevel hasValue(1)))
AtomComp(?x) ∧ hasConnector(?x, ?c)
  → hasInst(?x, ?c)
AtomComp(?x) ∧ hasConnector(?x, ?c)
    ∧ hasCompUnit(?c, ?cu) → hasInst(?x, ?cu)
```

A composite component on the other hand is one or more components composed together by a connector of arity equal to the number of components where the connector exists at level. For a composite component, its instances include the top-most composition connector together with the instances included in all-child components. It models as follows:

```
Class(CompositeComp partial Component
  restriction(hasConnector
    allValuesFrom(CompositionConnector)))
CompositeComp(?x) ∧ hasConnector(?x, ?c)
  ∧ hasLevel(?c, ?l) → hasLevel(?x, ?l)
CompositeComp(?x) ∧ hasConnector(?x, ?c)
→hasInst(?x, ?c)
CompositeComp(?x) ∧ hasConnector(?x, ?c)
  ∧ hasCompSeq(?c, ?s) ∧ allListContents(?s, ?co)
    ∧ hasInst(?co, ?i) → hasInst(?x, ?i)
```

Specific kinds of composite components can be extended from *CompositeComp*. For example, *NumSelectorComp* and *OpenPipeComp* denote the components whose connectors are *NumSelector* and *OpenPipe*, respectively

```
Class(NumSelectorComp partial CompositeComp
  restriction(hasConnector
   allValuesFrom(NumSelector)))
Class(OpenPipeComp partial CompositeComp
  restriction(hasConnector
   allValuesFrom(OpenPipe)))
```

3.4 Method execution

The exogenous-connector component model separates the control and computation, which makes components truly independent and therefore more reusable in different architectures. A *computation unit* performs computation within itself, and does not invoke computation in another unit. We model this as the class *ICInvocation*, and *invoMethod* denotes the method to be invoked. The last SWRL rule ensures that an *invocation connector* can only invoke a method that has been implemented by its computation unit.

```
ObjectProperty(fromConnector functional
  domain(ICInvocation) range(InvocationConnector))
ObjectProperty(invoMethod functional
  domain(ICInvocation) range(invoMethod))
Class(ICInvocation
  restriction(fromConnector cardinality(1))
  restriction(invoMethod cardinality(1)))
ICInvocation(?x) ∧ fromConnector(?x, ?ac) ∧
  invoMethod(?x, ?m) ∧ hasCompUnit(?ac, ?cu)
    → hasMethods(?cu, ?m)
```

The component execution is defined following the composition hierarchy, where methods to be executed are passed to a component in a tree structure (denoted by *MethodTree*). A *MethodTree* is either a single method or a sequence of *MethodTree*. *Execution*, representing a component execution, uses the property, *exeComp* and *exeMethodTree*, to denote the component where the execution is called and the methods are to be executed.

```
Class(MethodTree complete unionOf(Method MethodTreeSeq)
Class(MethodTreeSeq partial OWLList
  restriction(hasContents AllValuesFrom(MethodTree))
  restriction(isFollowedBy
   AllValuesFrom(MethodTreeSeq)))
  ObjectProperty(exeComp functional
   domain(Execution) range(Component))
  ObjectProperty(exeMethodTree functional
   domain(Execution) range(MethodTree))
  Class(Execution
  restriction(exeMethodTree cardinality(1))
  restriction(exeComp cardinality(1)))
```

The execution of an *atomic component* is defined as *AtomExe*. An *atomic component* can only execute a single method. The last SWRL rule ensures that the executed method is the same as the method, which the component's invocation connecter invoked. The above definition of

ICInvocation, in turn, ensures that the method is one of the methods provided by the computation unit.

```
Class(AtomExe partial Execution
  restriction(exeComp allValuesFrom(AtomComp))
  restriction(exeMethodTree allValuesFrom(Method)))
AtomExe(?ae) ∧ exeComp(?ae, ?aComp)∧
    exeMethodTree(?ae,?aMethod)∧
    hasConnector(?aComp, ?invConnector) ∧
    hasCompUnit(?invConnector, ?compUnit) ∧
    ICInvocation(?y) ∧ fromConnector(?y, ?invConnector)
    ∧ invoMethod(?y, ?aMethod1)
  → sameAs(?aMethod, ?aMethod1)
```

The execution of *numerical selector* components is denoted by the class, *NumExe*. The property, *selectInd*, is used to denote the child component to be executed. The last rule (lines 6–19) models the behavior of *NumExe*. Basically, it tries to find out the *?index*th component in the number selector connector and execute the method tree denoted by the variable, *seqMethodTree*. The execution of *pipe* components can be defined as well.

```
(1)  DatatypeProperty(selectInd functional
(2)    domain(NumExe) range(Integer))
(3)  Class(NumExe partial Execution
(4)    restriction(selectInd cardinality(1))
(5)    restriction(exeComp
         allValuesFrom(NumSelectorComp)))
(6)  NumExe(?ne) ∧ exeComp(?ne, ?nComp)
(7)    ∧ exeMethodTree(?ne, ?seqMethodTree) ∧
(8)    selectInd(?ne, ?index) ∧
(9)    hasConnector(?nComp, ?numConnector)∧
(10)     hasCompSeq(?numConnector, ?comSeq) ∧
(11)     size(?comSeq, ?si) ∧ swrlb:add(?si1,?si,1)
(12)     ∧ swrlb:add(?si1, ?position, ?index) ∧
(13)     isFollowedBy(?comSeq, ?indexedComSeq)∧
(14)     size(?indexedComSeq, ?position) ∧
(15)     hasContents(?indexedComSeq, ?indexedComp)
(16)     Execution(?compEx) ∧
(17)     exeComp(?compEx, ?indexedComp) ∧
(18)     exeMethodTree(?compEx, ?seqMethodTree1)
(19)       → sameAs(?seqMethodTree,?seqMethodTree1)
```

3.5 Non-architectural metadata

Non-architectural properties of a component system are at least as important as its somewhat more classical properties related to component composition. They must be considered as early as possible in the component development life cycle to avoid costly failures. Those non-architectural properties are important for both component developers and users. Component developers must implement components in such a way that they have determinable non-functional properties and application assemblers, and the runtime system must use these components so that the non-functional properties required from the application can be guaranteed.

To make this ontology of the component model more effective for tasks like component management, composition and deployment, we also develop the formal OWL ontology for these

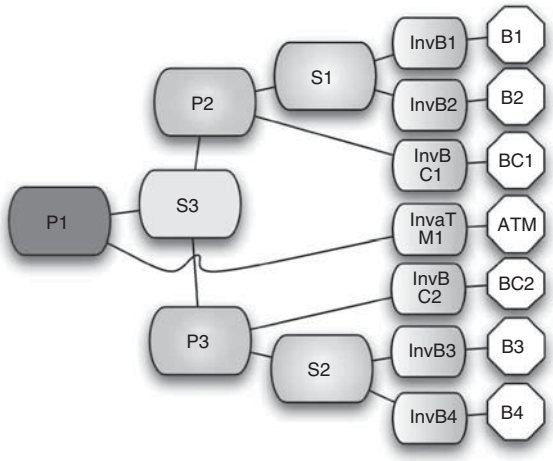

Figure 2 The bank system example: exogenous-connector architecture

non-architectural properties. These non-architectural metadata aim to provide at least three different aspects of information regarding a component as follows:

Component management properties such as *creator*, creating *date*, implementing *language* and *version number*, etc., for a component or connector.
Component understanding properties that provide human-readable information about the usage and functionality of entities within the component model.
Execution environment properties, which characterize the constraints that the components will execute at runtime, such as *platform* and *OS*.

3.6 System modeling

In this section, we show the usage of OWL for specifying a concrete system using the ontology we just defined. A simple bank system is used to illustrate the idea. This bank system has only one ATM that serves two bank consortia (BC1 and BC2), each with two bank branches (B1 and B2, B3 and B4, respectively). The ATM passes customer requests together with customer details to the customers' bank consortium, which in turn passes them on to the customers' bank branches. The bank branches provide the usual services of withdrawal, deposit, balance check, etc. Figure 2 (reproduced from Lau *et al.*, 2005) shows the connector architecture of the bank component application. More details about this bank system and its implementation can be found in Lau *et al.* (2005).

In this example, there are three kinds of *computation units*, that is, *Bank*, *BC* and *ATM*, which perform computations of bank branches, bank consortia and ATMs, respectively. Five different methods are offered by these computation units:

```
Individual(deposit type(Method))
Individual(getBalance type(Method))
Individual(withdraw type(Method))
Individual(locateBank type(Method))
Individual(locateBC type(Method))
Class(Bank partial ComputationUnit
  restriction(hasMethod
    allValuesFrom({getBalance deposit withdraw}))
  restriction(hasMethod somevalueFrom({getBalance}))
  restriction(hasMethod somevalueFrom({deposit}))
  restriction(hasMethod somevalueFrom({withdraw})))
Class(BC partial ComputationUnit
```

```
  restriction(hasMethod allValuesFrom({locateBank}))
  restriction(hasMethod somevalueFrom({locateBank})))
Class(ATM partial ComputationUnit
  restriction(hasMethod allValuesFrom({locateBC}))
  restriction(hasMethod somevalueFrom({locateBC})))
```

Next, we define the required computation unit instances. Let us assume that in this bank system there are four bank instances, two bank consortia instances and one ATM instance. Each of them controls a computation unit:

```
Individual(Bank1 type(Bank))
... ...
Individual(Bank4 type(Bank))
Individual(BC1 type(BC)) Individual(BC2 type(BC))
Individual(ATM1 type(ATM))
```

We will now define the first level of the composition hierarchy, i.e., the composition of invocation connectors and computation units to atomic components

```
Individual(InvB1 type(InvocationConnector)
  value(hasCompUnit Bank1))
... ...
Individual(InvB4 type(InvocationConnector)
  value(hasCompUnit Bank4))
Individual(InvBC1 type(InvocationConnector)
  value(hasCompUnit BC1))
Individual(InvBC2 type(InvocationConnector)
  value(hasCompUnit BC2))
Individual(InvATM1 type(InvocationConnector)
  value(hasCompUnit ATM1))
```

In the component model, the connector structure is constructed level by level. At level one, an invocation connector is connected to every atomic component. This enables all the methods of a component to be invoked

```
Individual(AB1 type(AtomComp)
  value(hasConnector InvB1))
... ...
Individual(ABC1 type(AtomComp)
  value(hasConnector InvBC1))
Individual(ABC2 type(AtomComp)
  value(hasConnector InvBC2))
Individual(AATM type(AtomComp)
  value(hasConnector InvATM1))
```

Then at Level 2, invocation connectors are connected by Level 2 connectors, which affect appropriate behavior among the components connected to the invocation connectors. In the bank example, there are two second-level composition connectors: S1, which is used to select the customer bank branch from banks B1 and B2 before invoking that branch's methods requested by the customer and S2, for choosing between B3 and B4 before invoking their methods requested by the customer. Two level-two composite components, CB12 and CB34, are created with respect to CB12 and CB34. The following shows the OWL model. Note that $\langle$AB1, AB2$\rangle$ represents the sequence of AB1 and AB2, modeled using the pattern mentioned in the earlier section

```
Individual(S1 type(NumSelector)
  value(hasLevel 2) value(hasCompSeq<AB1,AB2>))
```

```
Individual(S2 type(NumSelector)
  value(hasLevel 2) value(hasCompSeq<AB3,AB4>))
Individual(CB12 type(CompositeComp)
  value(hasConnector S1))
Individual(CB34 type(CompositeComp)
  value(hasConnector S2))
```

Similarly, we can model the rest of the system; at Level 3, we have two pipe connectors, P2 and P3, for BC1 and BC2, respectively. At Level 4, S3 is a selector connector that selects the customer's bank consortium from consortia BC1 and BC2. Finally, at Level 5, the pipe connector, P1, initiates the banking system's operational cycle by passing customer requests and card information to the ATM, invoking the ATM methods, and then passing the resulting value to connector S3

```
Individual(P2 type(OpenPipe)
  value(hasLevel 3) value(hasCompSeq<ABC1,CB12>))
Individual(P3 type(OpenPipe)
  value(hasLevel 3) value(hasCompSeq<ABC2,CB34>))
Individual(CBC1 type(CompositeComp)
  value(hasConnector P2))
Individual(CBC2 type(CompositeComp)
  value(hasConnector P3))
Individual(S3 type(NumSelector) value(hasLevel 4)
  value(hasCompSeq <CBC1,CBC2>))
Individual(CBCs type(CompositeComp)
  value(hasConnector S3))
Individual(P1 type(OpenPipe)
  value(hasLevel 5)
  value(hasCompSeq <AATM1,CBCs>))
Individual(BankSystem type(CompositeComp)
  value(hasConnector P1))
```

Note that the translation from a component configuration to the corresponding OWL model can be achieved automatically by tools. There are also some tools available for automatically or semi-automatically annotating components using ontology (Marinova *et al*., 2008).

4 Automated reasoning support on component models

4.1 Consistency checking

The OWL and SWRL reasoner can identify the inconsistency of a composition with full automation. For example, assume that the definition of the composite connector, P2, from the earlier bank example has been changed as follows:

```
Individual(P2 type(OpenPipe)
  value(hasLevel 3)
    value(hasCompSeq<ABC1,CB12, BankSystem>))
```

P2 intends to call a method implemented in ATM through connector P1, so that it connects three components—ABC1, CB12, and *BankSystem*. The Semantic Web reasoning engine will report an inconsistency. This is because we stated in a modeled fact that for a composite connector, it can only connect to components below its level; however, in this case, a Level 3 connector (P2) connects a Level 5 component (*BankSystem*). Moreover, we can also define a set of queries to help the users find the common errors that often occur in a composition. These query rules can provide handy debugging assistance to system developers. For example, the

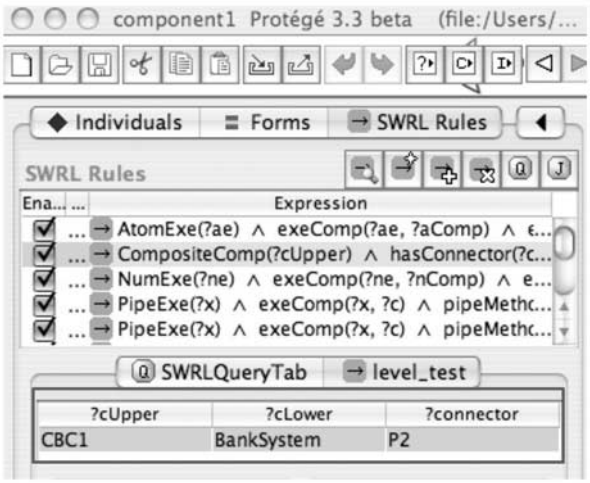

Figure 3 The consistency checking example

following query can spot the above error: Figure 3 shows the query result in the Protégé SWRL tab

```
CompositeComp(?cUpper) ∧
  hasConnector(?cUpper, ?connector) ∧
  hasCompSeq(?connector, ?cSeq) ∧
  allListContents(?cSeq, ?cLower) ∧
  hasLevel(?cUpper, ?levelUpper) ∧
  hasLevel(?cLower, ?levelLower) ∧
  swrlb:lessThanOrEqual(?levelUpper, ?levelLower)
→ query:select
(?cUpper, ?cLower, ?levelUpper, ?levelLower)
```

The above query shows another example of inconsistency checking, when we configure *BankSystem* to execute the method sequence $\langle locateBank, locateBC \rangle$. According to our model, the execution of the first-child component of *BankSystem* (connected by the connector P1), that is, AATM1, must execute the first element in the method sequence, which is *locateBank*, in this case. However, the computation unit (ATM) connected to the AATM1's connector, *InvATM1*, does not implement the method, *locateBank*. Thus, the correct method sequence should be $\langle locateBC, locateBank \rangle$.

With the growth in the number of components, connectors, and the complication of compositions in a component model, manual checking of the consistency of a configuration is very laborious and highly error-prone. As ontology-reasoning tools are developed to reason about knowledge bases with enormous size, our approach has a significant advantage in the verification of huge component models.

4.2 Model querying

One of the major contributions of the OWL knowledge model introduced by the Semantic Web community is that it allows the users to perform more accurate and meaningful searches. This strength of OWL can be applied in the component context as well. Useful OWL queries can be formulated for assisting component management and choosing the most suitable candidate components for composition.

There are many OWL query systems available or under development. In this paper, the Jess engine and Protégé SWRL query tab are used to show the types of queries that can be achieved in

the proposed semantic environment. For example, a system developer has a component *C* and s/he wants to find out all the atomic components that have the same execution platform as *C*, implement the method *M* and have 'OO-based' syntax. The syntax of components can be divided into three different categories, i.e., OO programing languages-based, programming languages with IDL mappings-based and ADL-based syntax (Lau & Wang, 2007). The following query retrieves the result (the JESS SWRL query syntax is used here), where *executionEnv* and *hasLanguage* are OWL properties used to denote the execution environment (such as 'Web' or 'Desktop') that a component can be deployed to and its implementation language. The *OOComponent* is an OWL *defined* class (denoted by the keyword *complete*), which represents the atomic components, the connectors of which connect to only those computation units implemented by 'JavaBeans' or 'EJB'. 'JavaBeans' and 'EJB' are the only two major component systems with OO-based syntax (Lau & Wang, 2007). Note that when we add a component to the repository, we do not explicitly assert whether it is an *OOComponent* or not. Using the necessary and sufficient definition of *OOComponent*, the OWL reasoner can automatically infer these relationships. By doing so, it can make the component repository cleaner and easy to maintain. As we can normalize the complicated multi-axial poly-hierarchical classification of components to simple non-overlapping trees, the relationships among defined concepts are maintained by the classifier. For presentation purpose, we explicitly define the named class *OOComponent*. In practice, OWL anonymous class can be used directly in the query

```
AtomComp(C) ∧ executionEnv(C,?e)
  ∧ hasConnector(C,?con)
  ∧ hasMethods(?con, m) ∧ OOComponent(?x)
∧ executionEnv(?x,?e) →query:select(?x)
Class(OOComponent complete AtomComp
  restriction(hasConnector
    allValuesFrom(intersectionOf(InvocationConnector
      restriction(hasCompUnit
        allVauesFrom(intersectionOf(ComputationUnit
          restriction(hasLanguage
          someValuesFrom({JavaBeans EJB}))))))))
```

Apart from the checking and querying of component configuration consistency, OWL reasoners can also be used to infer the subsumption relationships between two component configurations, if a configuration satisfies all the constraints from another configuration. Furthermore, the ontology we develop can provide a standard method to annotate components and assist the CBD.

5 Related works and conclusion

There are a few related works. Researchers recently investigated how Semantic Web and SWSs can be used to build a flexible environment for supporting, extending and integrating various formal specification languages (Dong *et al.*, 2002). One additional benefit is that OWL and RDF query techniques can facilitate formal specification comprehension. Oberle *et al.* (2006) developed a framework of ontologies that formalize modularization and communication in large software systems.

The exogenous-connector component model could be considered as a prominent improvement of the existing CBD. It allows the users to separate control from computation in the design phase, which makes components truly independent and therefore more reusable in different architectures. Current efforts on exogenous-connector component modeling are largely graphical and informal, which could hinder the precise representation and automated analysis of component modeling. Component composition verification is an important task in CDB. With the growth in the number of components in a software application, manual checking of validity is very laborious and error-prone. In this paper, we propose a Semantic Web approach to component modeling and verification. We use the OWL+SWRL ontology language to represent exogenous connectors

and components in an unambiguous manner. As OWL+SWRL has a formal and rigorous semantical basis and because of the decidability of OWL DL, fully automated analysis is achievable. In our approach, we use OWL+SWRL reasoning engines, such as Jess and Racer, to perform automated verification over the OWL representation of the component models. This analysis helps us to detect possible inconsistencies in component composition. As reasoning engines are designed to handle large-scale knowledge bases, an efficient and effective analysis of large component models is made possible. We used the bank example, a standard problem for evaluating CBD technologies, in the paper to illustrate our approach. We showed that inconsistencies can be effectively detected by the reasoning engines, and queries can be performed to comprehend the desired properties of the model. We believe that the semantic web can play important roles in CBD, and we will continue to explore the synergies between them.

We do realize that there is a limitation on the expressiveness of current Semantic Web languages. In this paper, we mainly focused on using ontology to represent and verify the structure of component composition. Lau *et al*. (2006a) presented a FOL semantic of the component model, which can be used to verify more complicated properties such as a component's functionalities (i.e., *preconditions* and *effects* of a component execution). However, verifying the complicated properties requires the user's interaction. Here we do not claim that OWL+SWRL is the only and best formal tool to reason over component models, but we do claim that it is an effective attempt with certain novel and irreplaceable advantages such as full automation and scalability. In fact, it is unlikely in the near future that both expressive and automatic tools will be developed.

Owing to the lack of large component systems that are publicly available, our current evaluation is based mainly on some relatively small or manually generated component models. In the future, we will evaluate our approach using more complicated real-life component systems. We are also developing a visual CASE tool to facilitate the creation and reasoning of component models.

References

Brickley, D. & Guha, R. V. (eds). 2004. *Resource Description Framework (RDF) Schema Specification 1.0*. `http://www.w3.org/TR/rdf-schema/`

Dong, J. S., Sun, J. & Wang, H. 2002. Semantic web for extending and linking formalisms. In *Proceedings of Formal Methods Europe: FME'02*, Eriksson, L.-H. & Lindsay, P. A. (eds). Springer-Verlag.

Drummond, N., Rector, A. L., Stevens, R., Moulton, G., Horridge, M., Wang, H. & Seidenberg, J. 2006. *Putting OWL in Order: Patterns for Sequences in OWL*, in 2nd OWL Experiences and Directions Workshop, Athens, GA.

Evren Sirin, B. P. 2004. Pellet: An owl dl reasoner. In *Proceedings of the International Workshop on Description Logics (DL2004)*, Volker Haaslev, R. M. (ed.). Whistler, Canada.

Gennari, J., Musen, M. A., Fergerson, R. W., Grosso, W. E., Crubézy, M., Eriksson, H., Noy, N. F. & Tu, S. W. 2002. *The Evolution of Protégé: An Environment for Knowledge-based Systems Development*. Technical report SMI-2002-0943, Stanford Medical Informatics, Stanford University.

Haarslev, V. & Möller, R. 2001. RACER system description. In *Proceedings of Automated Reasoning: First International Joint Conference*, Lecture Notes in Computer Science **2083**, 701–706. Siena, Italy.

Horrocks, I. n.d. *Fact++ web site*, `http://owl.man.ac.uk/factplusplus/`

Horrocks, I., Patel-Schneider, P., Boley, H., Tabet, S., Grosof, B. & Dean, M. 2003. *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*. `http://www.daml.org/2003/11/swrl/`

Knublauch, H., Fergerson, R. W., Noy, N. F. & Musen, M. A. 2004. The Protégé OWL plugin: an open development environment for Semantic Web applications. In *Proceedings of the 3rd International Semantic Web Conference (ISWC 2004)*. Hiroshima, Japan.

Laboratories, S. N. 1991. *Jess, the Rule Engine for the Java Platform*. `http://herzberg.ca.sandia.gov/jess`

Lassila, O. (eds), R. R. S. 1999. *Resource Description Framework (RDF) Model and Syntax Specification*. `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/`

Lau, K.-K. & Wang, Z. 2007. Software component models. *IEEE Transactions on Software Engineering* **33**(10), 709–724.

Lau, K.-K., Velasco Elizondo, P. & Wang, Z. 2005. Exogenous connectors for software components. In *Proceedings of 8th International SIGSOFT Symposium on Component-based Software Engineering*, Lecture Notes in Computer Science **3489**, 90–106.

Lau, K.-K., Ornaghi, M. & Wang, Z. 2006a. A software component model and its preliminary formalisation. In *Proceedings of 4th International Symposium on Formal Methods for Components and Objects*, de Boer, F. *et al.* (eds), Lecture Notes in Computer Science **4111**, 1–21. Springer-Verlag.

Lau, K.-K., Wang, Z., Wang, A. & Gu, M. 2006b. A component-based approach to verified software: what, why, how and what next? In *Proceedings of 1st Asian Working Conference on Verified Software*, 225–229. Chen, X., Liu, Z. & Reed, M. (eds). UNU-IIST report no. 347.

Marinova, Z., Amardeilh, F., Georgiev, K. & Francart, T. 2008. *Transitioning Applications to Ontologies: User Tools*. Technical report D3.4, TAO Project Deliverable. `http://www.tao-project.eu/resources/publicdeliverables/d3-4-1-final.pdf`

McGuinness, D. L. & van Harmelen, F. (eds). 2003. *OWL Web Ontology Language Overview*. `http://www.w3.org/TR/2003/PR-owl-features-20031215/`

Meyer, B. 2003. The grand challenge of trusted components. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. IEEE Computer Society, 660–667.

Oberle, D., Lamparter, S., Grimm, S., Vrandecic, D., Staab, S. & Gangemi, A. 2006. Towards ontologies for formalizing modularization and communication in large software systems. *Applied Ontology* **1**(2), 163–202.

Shaw, M. & Garlan, D. 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall.

Szyperski, C. 1998. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley.