University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Formal Specification and Practical Verification of Contextual Component Compatibility

Premek Brada

Technical Report No. DCSE/TR-2011-06
December, 2011

Distribution: public

Technical Report No. DCSE/TR-2011-06
December 2011

Formal Specification and Practical Verification of Contextual Component Compatibility

Premek Brada

Abstract

In this report we formally describe and practically verify a method which ensures run-time type consistency of component-based applications, by performing type-based substitutability checks as part of the component binding and update processes. The method takes into account the environment of the currently deployed component version and uses its so-called contextual complement in the checks. This novel approach overcomes the limitations of the standard notion of compatibility by allowing non-contravariant differences on the required side of the component's surface. The verification of the method uses an implementation for the OSGi component framework together with a simple GUI tool to perform component substitution experiments.

This work was supported by the Czech Science Foundation under the grant number P103/11/1489 "Methods of development and verification of component-based applications using natural language specifications".

Copies of this report are available on
http://www.kiv.zcu.cz/publications/
or by surface mail on request sent to the following address:

University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Contents

1 Introduction 2

1.1 Goals and Structure of this Paper 2

2 Related Work 3

3 Component Type and Deployment Context 4

3.1 Type Representation of Component Interface 4

3.2 Component Context 6

4 Component Substitutability 9

4.1 Contextual Substitutability 10

5 Realization and Experimental Tool for the OSGi framework 12

5.1 Architecture of the Tool Set 12

5.2 Key Methods and Algorithms 13

5.3 Tool for Experimental Verification of Contextual Substitutability . 14

6 Conclusion 15

1 Introduction

Component frameworks like Spring or OSGi [18] are increasingly often used to develop software systems with complex architectures, despite their relative simplicity in terms of the underlying component models.

Even though it is easier to manage complex architectures with components, safe evolution of these architectures — and preserving the internal consistency of the application throughout the evolution in particular — is still a challenging task. This is especially true for dynamic architectures (those which can evolve during application run-time) where changes cannot be fully anticipated by the architects and are not necessarily governed by any rules or patterns.

In our work we address the common scenario in which consistency verification is needed — the dynamic replacement of a component currently deployed in an architecture by another component, be it a completely different one (leading to the verification of *substitutability*) or a new version of the current one (resulting in the verification of *backward compatibility* as a special case). The verification is governed by the general principle of substitutability, summarised by Wegner and Zdonik [25]: *a replacement component should be usable whenever the current one was expected, without the client noticing it.*

Relevant verification methods vary but in the general case of an "open world scenario" where there need not be any relation between the current and replacement components, we need to check substitutability in-situ for the pair of components or component sets under substitution.

1.1 Goals and Structure of this Paper

Research approaches to safe substitution aim to ensure reliable substitutability by employing formal methods. However, the models used in these approaches tend to be too complicated to be usable by average software developers and the methods often suffer from prohibitive algorithmic or space complexity (cf. for example [15, 16]).

Industrial systems on the other hand almost exclusively use rather simple meta-data, most often version identifiers (e.g. [18]), to manually tag components as being compatible with their previous versions. The key disadvantage of this approach is fragility caused by the reliance on human effort to provide correct meta-data.

We have therefore developed a practically-oriented method of verifying substitutability of black-box components. It uses the (sub)type relation as its foundation since its formal strength is sufficient to prevent serious run-time errors while its evaluation can be done on current state of the practice component models, with relative simplicity and low algorithmic complexity.

The method is only as strong in terms of formal verification of component's substitutability as the component contract specification at hand allows. Mostly, it will therefore offer type-safety guarantees for current industrial component frameworks but it is able to incorporate the assessment of extra-functional properties compatibility [14] or advanced formal checking (e.g. to assert behavioural protocol compliance [19]) where appropriate data are available as part of the component's specification.

A novel and significant aspect of the method is the possibility to overcome the limits of standard subtyping — strict covariance of provided and contravariance of required features — by considering the environment in which the components are deployed (e.g. the application architecture or component framework's container), hereafter termed the *deployment context*.

The report is structured as follows. The following section provides an overview of the state of the art in component substitutability. Section 3 contains foundational formal definitions of component type and deployment context, and section 4 defines the notion of contextual substitutability.

The second part of the paper contains validation of the formal part. In section 5 we describe an implementation of the method for the OSGi component framework and a tool to perform a set of experiments using XML configuration and a GUI front-end.

2 Related Work

This work contributes to the challenging area of correctness and robustness in component frameworks "in the absence of a closed-world assumption" [22]. This is still a relevant issue, since for example Taylor [23] notes that in service-oriented architecture research there has been "little attention to orchestrating [architectural] changes across service (. . .) boundaries".

On a very high abstraction level, Georgas et al. [12] use a model of application architecture at run-time to manage its evolution. Constraints can be specified on the policies governing the evolution (adaptation) in order to preserve chosen architectural properties. The work however does not provide concrete details about the model, the constraints and ways to check them.

Many research approaches have addressed this need using holistic approaches with global integrity properties [21, 11]. Chaki et al. [9] for example use compositional reasoning and dynamic assume-guarantee checks to provide formally sound evaluation of substitutability with similar practical properties as our contextual one.

Most of these methods are however based on advanced formal systems (e.g. model checking, behavioural subtyping) often supported by specialized specification notations. These methods tend to suffer from prohibitive algorithmic or state complexity [16, 9] and the notations tend to be too complicated to be usable by average software developers [15].

Few research works have been concerned with use in industrial component frameworks. Polakovic et al. [20] implement architectural consistency checks for a resource-constrained component model, using a combination of compile-time type conformance verification and error handling code. Our approach would be hardly feasible in such cases due to the resource demands.

The work closest to ours in its spirit is Belguidoum and Dagnat's [3] which extends an earlier version of our contextual substitutability [5] with the notion of forbidden dependencies. This prevents multiple conflicting implementations of the same component feature (e.g. a messaging service) to invalidate context invariants. While this is an important observation, the substitutability itself is formalized on a rather abstract level and further refinement of service comparison methods, effects of component (de)installation, and forbidden dependencies representation is needed for a full applicability of the approach.

Last but not least, several methods that use type systems have been proposed [11, 17]. They moreover enable multi-component substitution which goes beyond our method proposed in this paper. However, their approaches have not been validated on industrial component frameworks.

3 Component Type and Deployment Context

To provide a sound basis for type-based substitutability and compatibility verification, a formal model at the type level of the compared components has to be available. In this section we describe such a model together with the representation of component's view of its deployment context.

3.1 Type Representation of Component Interface

A component implementation, available in some form of distribution package, may comprise many different elements — executable code, meta-data, resource objects, etc. All elements that are part of the component's interface (i.e. are accessible on the surface of the component's black box) may participate in inter-component interactions — both the provided ones, used by component's clients, and required ones, which express component's dependencies that need to be satisfied in order to guarantee the provided functionality or properties.

We capture the structure of the component interface in the form of *component type*, a structured data type [8] which contains all such elements and their role with respect to inter-component interactions. The information about the component and its interface elements is assumed to be available from some kind of specification; the internals of the implementation are abstracted from since the component is a black box.

Definition 1 *A* component interface element *is a tuple $e = (n, T, r, o, a)$ where $n \in String \cup \{\epsilon\}$ is the element's name (possibly empty), T is element's type (in the respective specification language type system), $r \in \{provided, required\}$ is the element's role on the component interface, $o \in Boolean$ is an indication whether the presence of the element at run time is optional, and $a \in N \cup \{*\}$ denotes arity (how many counterpart elements can be bound to it; '*' stands for "any").*

Examples of component interface elements are: a named OSGi service typed to an interface or class, a Java interface implemented by a EJB component (in this case $e.n = \epsilon$)[1], or property with a primitive type of a SOFA component. Optionality of the element can be indicated directly in the component specification (as in OSGi "optional" directive on imported packages or similar Fractal attribute of a component's interface) or via element cardinality.

Definition 2 *Let $E_C = \{e_i\}_{i \in I}$ (for a finite index set I) be the set of all component interface elements of a* component implementation **C** *which can be observed from outside of its black box.*

The component type *C of* **C** *is a pair of provided and required element sets: $C = (E^P, E^R) \mid (E^P \cup E^R = E_C) \wedge (E^P \cap E^R = \emptyset)$ where $(\forall e \in E^P : e.r = provided)$ $\wedge$ $(\forall e \in E^R : e.r = required)$.*

The component *as a run-time instance of* **C** *is a tuple $c = (id, C, P, R)$ where id is a unique identification of the instance, C its component type, P and R are the sets of interface elements actually present at the instance's interface. (We will abbreviate the expression "component* **C** *with type C" to just "component C" where unambiguous.)*

The element role (provided, required) is distinguished in the component type for a fundamental reason: it affects handling of component during type operations, namely matching and subtype comparison. In this respect this definition of component type and its associated (sub)typing rules, while similar to standard structured data types, reflect the core notion of component-based programming.

[1] Notational remark: Throughout the text, we use the dot notation to denote the individual parts of a tuple, so for $A = (a, b)$ the expression $A.a$ denotes the first part of A's structure.

The latter two parts of the component's tuple represent its effective type at the given time. It holds that $c.P \subseteq c.C.E^P$ and $c.R \subseteq c.C.E^R$ because (some of) the optional elements may be omitted by the instance. We note further that the set of c's interface elements may change in time, i.e. $\exists t_1 \neq t_2 \cdot c.P/t_1 \neq c.P/t_2$ and likewise with $c.R$.

The subtype relation for interface elements is the standard reflexive transitive one, so it is a preorder on types:

Definition 3 *We say that component interface element e_i is a* subtype of element e_j *(denoted $e_i <: e_j$) if all of the following conditions hold:*

- $e_i.T <: e_j.T$ *(the element types are in a subtyping relation);*
- $e_i.r = e_j.r$*;*
- $\begin{cases} \neg e_j.o \Rightarrow e_i.o = \mathit{false} & \text{if } e_i.r = \mathit{provided}, \\ e_j.o \Rightarrow e_i.o = \mathit{true} & \text{if } e_i.r = \mathit{required}; \end{cases}$
- $\begin{cases} e_i.a \geq e_j.a & \text{if } e_i.r = \mathit{provided}, \\ e_i.a \leq e_j.a & \text{if } e_i.r = \mathit{required}. \end{cases}$

The next subsection formalizes the representation of the environment in which a component is deployed.

3.2 Component Context

We noted in the introduction that it is useful to capture the deployment environment of a particular component for evaluating substitutability. This *component deployment context* contains the other components and architectural connections within the environment in which the component is employed. The environment can be a component cluster (a closely coupled part of a component application), the component-based application or the whole run-time environment surrounding a deployed component in the run-time framework.

Definition 4 *The* deployment context *of a component instance c is a tuple $D = (K, B)$ where $K = \{c_i\}_{i \in I}$, $c \in K$ is the set of components existing in the deployment environment of c, and $B = \{(e^p, e^r) \mid \exists C_p, C_r \in K \cdot (C_p \neq C_r) \wedge (e^p \in C_p.P) \wedge (e^r \in C_r.R) \wedge e^r$ is bound to $e^p\}$ are the bindings between component elements within the context.*

A deployment context D is architecturally consistent *if all the inter-component bindings are correctly resolved (including the matching of types of bound elements) and the components are correctly functioning.*

A subset of this environment which is particularly interesting from the substitutability point of view is a component's complement within the deployment context. The model of the complement uses the same abstractions as that of the component type described in the previous subsection.

Definition 5 *Assume an architecturally consistent context D and component $c \in D.K$ with component type C. The* contextual complement of c in D *is a "virtual" component type $\bar{C}_D = (\bar{P}, \bar{R})$ such that*

- $\bar{P} = \{e \mid (\exists c_r \in D.K \cdot c_r \neq c, e \in c_r.R) \wedge (\exists e_p \in c.P \cdot (e_p, e) \in D.B)\}$ – *this set consists of the actual client elements of c's provided ones.*
- $\bar{R} = \bigcup_s c_s.P \mid c_s \in D.K, c_s \neq c$ – *this set consists of all elements available in D which can satisfy a component's requirements;*

The complement can be seen as an inverted effective type of c at the given time (cf. the P, R element sets of the component instance). It captures the following two aspects of the context from the point of view of the component:

1. The real usage of the component's provided elements, as given by the bindings to particular required elements of other components.
2. The elements available in the environment that *can* possibly satisfy any c's substitute component's requirements, most commonly via other component's provided elements.

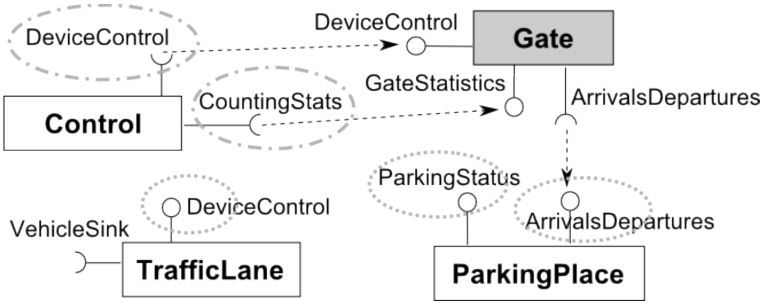


Figure 1: Component in an architecture and the elements forming its contextual complement

Several interesting observations can be made about the contextual complement from typing perspective. They are summed up in the following lemma.

Lemma 6 (Properties of contextual complement) *Assume a component c with type $C = (P, R)$ and its contextual complement $\bar{C}_D = (\bar{P}, \bar{R})$ according to the definitions above. Let $P' = \{e' \mid (e' \in P) \wedge (\exists \bar{e} \in \bar{P} \cdot (e', \bar{e}) \in D.B)\}$ be the set of* actually bound provisions *of C. Then it holds that*

1. *$P' \subseteq P$ (not all provisions need to be bound).*

2. *$\forall \bar{e} \in \bar{P}, e \in P \cdot (e, \bar{e}) \in D.B : e.T <: \bar{e}.T$ (context's bound required elements can have generalized types).*

3. *$\forall e \in R \cdot e.o = false$ $\exists \bar{R}_e \subseteq \bar{R}$ such that $\bar{R}_e \neq \emptyset \wedge \forall \bar{e} \in \bar{R}_e : \bar{e}.T <: e.T$ (all component's mandatory requirements are satisfiable by the context, via one or more elements with possibly specialized types).*

Proof. The proof of these properties is straightforward:

1. If all $e \in c.P$ are bound to client elements in the context, then $P' = P$; otherwise, $|P| - |P'|$ is the number of unbound provided elements.

2. Assume there is a required element $\bar{e}_x$ of some component $c_x \in D.K$ bound to a provided element e of c (consequently $\bar{e}_x \in \bar{R}$) such that $\bar{e}_x.T <: e.T$. Then the binding of the provided e to the other component's required $\bar{e}_x$ would be type-unsafe, since e cannot cover all the type features of $\bar{e}_x$. This violates the assumption of architectural consistency of the deployment context. (In practice, the run-time framework would decline to establish such binding a-priori.)

3. (a) Assume $\bar{P}_e = \emptyset$, meaning the non-optional required element e isn't bound to any corresponding provider. This again violates the architectural consistency assumption. (b) Assume $e.T <: \bar{e}.T$ which is a situation analogical to point 2. The type unsafety of the binding between e and $\bar{e}$ would eventually lead to a malfunction of c, which contradicts the assumptions (with the same practical interpretation).

■

Note finally that the $\bar{P}$ captures the real types of elements bound to the actually used $c.P$ elements, not simply the types of these, and that there is no requirement for element name uniqueness in the $\bar{P}$, $\bar{R}$ sets.

The idea of component's complement and its properties is illustrated on Figure 1. Component *Gate* is deployed in a simple architecture, bound to the *Control* and *ParkingPlace* components. An additional *TrafficLane* component is installed in the run-time framework. The deployment context of *Gate* comprises the two counterparts of its provided interfaces in the $\bar{R}$ set, and the three other provided interfaces available in the framework as its $\bar{P}$ set.

4 Component Substitutability

The principle of substitutability introduced in Section 1 provides a very general definition of the notion. In the context of component-based software engineering, it can be elaborated upon by taking into account the features available in the current state-of-the-art component models.

The replacement component is substitutable for a component currently deployed in a consistent application architecture if it satisfies the following general requirements [4]:

1. Presents the same operational interface (at the syntactic and typing level) to its environment.
2. Exchanges the same data (with respect to their location and format) as the current one.
3. Conforms to the semantics and behavioural specifications of the current component in all interactions in which it is engaged.
4. Exhibits compatible extra-functional (quality of service) characteristics.

In the approach presented here we concentrate on the first area, which is a deliberate simplification of the issue. The rationale for this decision is based on the challenges faced when working with industrial component frameworks. There, specifications of advanced aspects are not available or cannot be reconstructed from implementation in most cases; therefore, especially semantic compatibility is hard to verify.

We therefore need to base the formal notion of substitutability only on such artifacts and abstractions that are products of standard component development process. This lead us to (a) working with information directly available in the component distribution package — semi-formal component interface specifications, possible meta-data created during development, and data extracted by run-time component introspection; (b) using the least common denominator of the formal foundations — the type system and its subtyping rules — which are always available for any programming or specification language and provide a reasonable degree of trust in the conclusions as to the run-time safety of substitute component.

Let us now define the basic kind of type-based substitutability (presented in earlier versions in [6]). The following section then presents the novel kind of substitutability which considers the deployment context.

Definition 7 *We say that a (replacement) component type $R = (P', R')$ is* strictly substitutable *for the (current) component type $C = (P, R)$ if $(\forall e \in$*

P $\exists e' \in P' : e'.n = e.n \wedge e' <: e) \wedge (\forall e' \in R'\ \exists e \in R : e'.n = e.n \wedge e <: e')$. This fact is denoted as $R \prec C$.

The definition corresponds to the natural understanding of "vertical" compatibility [3]: the replacement component provides at least the same, and requires at most the same, component interface elements with respect to their names and types (irrespective of element's optionality). It uses the common notion of co- and contra-variance at the component type level (cf. [24] or [7]). This definition ensures a-priori substitutability of any pair of a component instance and its replacement which have the types C and R.

4.1 Contextual Substitutability

The principle of substitutability tells us that this property does not concern just the two components (current, replacement) in question: we also need to take into account their use by clients. From this point of view, changes in the provided and required parts of component interface do not affect substitutability equally.

In many component-based architectures not all of the component's provided features are utilised, i.e. bound to clients. On a case-by-case basis, these unused features can therefore be omitted when evaluating substitutability in the given deployment context. Similarly, it is common that new features are added during component evolution which results in the need to add corresponding dependencies to make them work. In the programming language research this led to the notion of covariance. In the case of deployed software components, we can take the advantage of the knowledge of deployment context and match the replacement component's extended requirements with any of those provided by the context's components.

From the architectural point of view this is a clean solution since the component should be agnostic of who is providing the required functionality, as long as it conforms to its stated specification. This leads to the following definition:

Definition 8 *Given a currently deployed component c with type C and its contextual complement $\bar{C}_D$, we say that the (replacement) component type $R = (P', R')$ is* contextually substitutable *for C if it holds that $R \prec \bar{C}_D$ that is if $(\forall \bar{e} \in \bar{P}$ $\exists e' \in P' : e' <: \bar{e}) \wedge (\forall e' \in R'\ \exists \bar{e} \in \bar{R} : \bar{e} <: e')$. This is denoted $R \prec_D C$.*

In plain words, the contextually substitutable replacement component provides at least the same features as are those used by the clients of the current one, and requires at most what is available in the context. It can be said it is horizontally compatible [3] with the context.

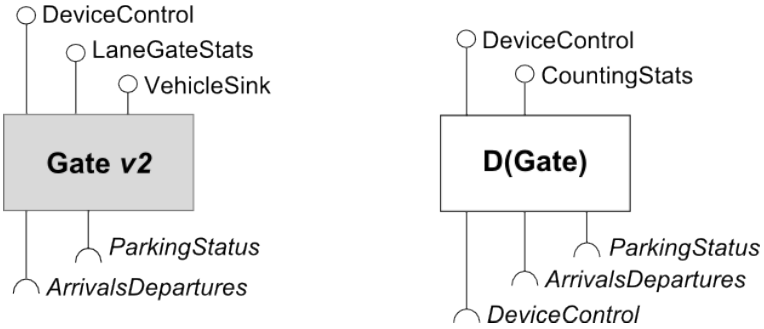


Figure 2: New component version vs. the contextual complement of the currently deployed one

Continuing with the example introduced in the previous section, Figure 2 shows a second version of the *Gate* component that should replace the original one. Its requirements are clearly greater than those of the original version, moreover one of the provided interfaces has changed its type. However, comparing *Gate v2* to the contextual complement of its currently deployed version shows that it can actually be used in the given architecture (provided the `LaneGateStats` interface is a subtype of `CountingStats`).

Intuitively, one would expect that strict substitutability implies contextual.

Proposition 9 (Strict substitutability implies contextual) *Assume components c and r with types C and R respectively. It holds that $\forall D \cdot c \in D.K : R \prec C \Rightarrow R \prec_D C$, i.e. if R is strictly substitutable for C then it is also contextually substitutable for C in any architecturally consistent deployment context D.*

Proof.

We first need to prove that $\forall D : C \prec_D \bar{C}_D$ meaning that C "fits in" its (any) context. Using the standard notation $C = (P, R)$ and $\bar{C}_D = (\bar{P}, \bar{R})$ this can be done in two parts:

- $\forall \bar{e} \in \bar{P}\ \exists e \in P : e <: \bar{e}$ (P is a substitute for $\bar{P}$) – follows from Lemma 6 items 1 and 2.
- $\forall e \in R\ \exists \bar{e} \in \bar{R} : \bar{e} <: e$ (R is a "supertype of" $\bar{R}$) – follows from Definition 4 ($|R| \leq |\bar{R}|$) and Lemma 6 item 3.

From the assumptions of the proposition we have $R \prec C$, the above says that effectively $C \prec \bar{C}_D$ and since the substitutability relation is transitive, it follows that $R \prec \bar{C}_D$.

■

This fact can be useful in certain common scenarios, e.g. in the special case of component *backward compatibility*: for a subsequent revision of a component we can easily prove strict substitutability with its immediatelly preceding revision at component release, store appropriate indication in its meta-data, and use it when upgrading the component.

Only if no such indication is available the assessment of substitutability must be carried out at the component binding or upgrade time. At this time it also makes sense to perform the contextual substitutability checks.

Due to its time-dependent nature, both the component's effective type (the actually used provided and required elements) and the deployment context may change in component instance's lifetime. Once compatibility is verified during upgrade, architectural consistency needs therefore to be continuously verified and ensured by other means.

5 Realization and Experimental Tool for the OSGi framework

Building on our previous work [6] we have implemented a contextual substitutability verifier for the OSGi framework [18] and a top-level GUI tool that helps to simulate various situations in which the contextual compatibility can be checked.

5.1 Architecture of the Tool Set

The overall technical design of the verifier was driven by several goals motivated by the need for practical utility. One of them was a simplified scope (evaluate compatibility of subsequent bundle[2] revisions, not any-to-any substitutability checks), another one a non-intrusive integration in the host framework.

The verifier application has the form of a set of OSGi bundles. The overall architecture of the implementation comprises three layers — a simple user interface, bundle and context representation loaders plus substitutability verifier (comparator), and an underlying Java type system model and subtyping rules implementation.

The first two layers are shown in Figure 3. Once the type representations for both the contextual complement and the replacement bundle are created by the *Loader* bundles (forming a tree data structure with provisions for primitive types

[2] Bundle is the OSGi term for a component.

and circular references), they are submitted to the *Bundle Comparator* which implements the substitutability verification. The result of its work is essentially an annotated type tree. It is aggregated into a single assertion about the type relation between the bundle and the complement. The *Substitution Verifier* bundle wraps the whole process, taking care of activating the loaders and comparator, and interpreting its result for the user.

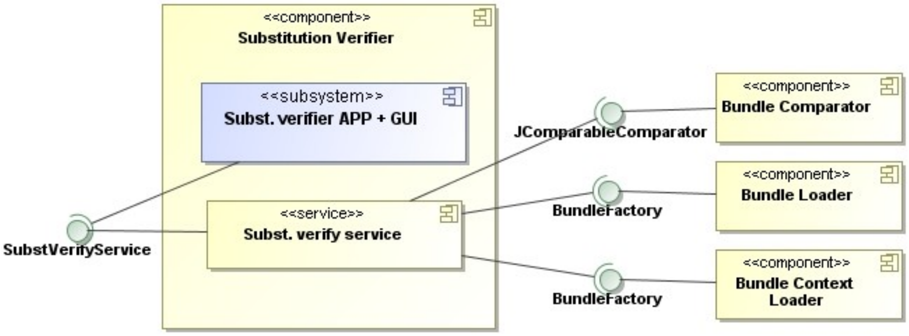


Figure 3: Architecture of the OSGi bundle compatibility verifier

Since no suitable run-time type representation of OSGi bundles is available, we use a custom-build model [1] called *BundleTypes*. It consists of domain classes capturing selected characteristics of the whole bundle, both at the module layer (its exported and imported packages) and the service layer (provided and depended-on services). This representation then references a lower layer model, called *JavaTypes*, which captures the type information of the individual Java classes.

5.2 Key Methods and Algorithms

The reconstruction of the type representations uses different means depending on the bundle in question. For the replacement bundle we use bytecode analysis with the ASM library[3] wrapped by a custom classloader and create stubs for shared (JRE, OSGi core, . . .) or unreachable classes, because the replacement bundle is accessible only as standalone .jar file. For the imported packages in particular we have to assemble their type representation [2] from class and operation references extracted from the bundle bytecode.

The type representation of the current bundle and its contextual complement can be obtained through standard OSGi framework services (package admin, bundle metadata and classloader methods) and Java reflection API. This information is easily reachable since the bundle's metadata and bindings to client and provider components are available in the respective framework registries.

[3] http://asm.ow2.org/

Concerning subtype relation implementation, the design of the algorithms had to reconcile the differences between the theoretical notion of the type relation — as used in the previous section — and the rules employed by Java as the actual specification language, its linking mechanism and the run-time system of the OSGi framework.

Most prominently, Java uses subclassing rather than subtyping [10] in its type matching rules and differentiates subtyping from binary compatibility [13]. This relation is actually the source of the underlying element "subtype" relation since OSGi bundles are bound and updated as binary .jar files.

Although OSGi is rich in features and modifiers at the module layer (optional imports, "uses" constraint, version ranges, etc.) most of them do not affect substitutability on the exported side and the effective type on the imported side (which reflects the effects of these modifiers) is easily obtained. At the service layer, element substitutability is verified implicitly — all service interfaces must be declared in the exported/imported packages so their type comparison is handled at the module layer.

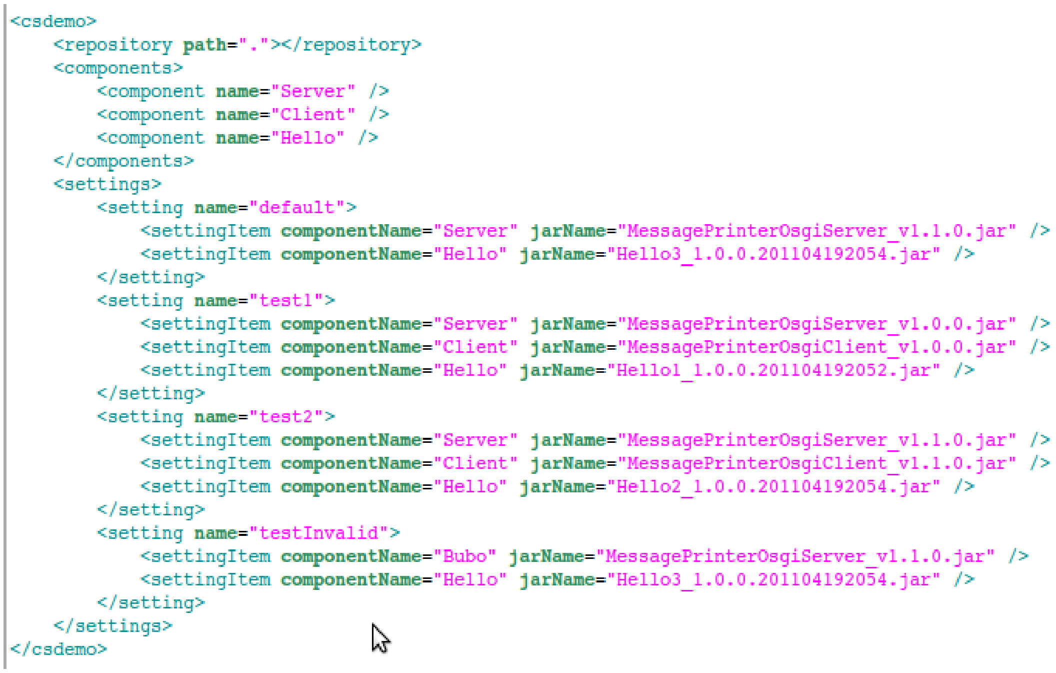
5.3 Tool for Experimental Verification of Contextual Substitutability

A simple tool has been created to enable experiments with contextual substitutability on OSGi bundles. It's purpose is to load an application configuration (consisting of a set of bundles), starting a pre-defined subset of bundles, and enable the user to observe the changes in substitutability of individual bundles as he or she interactively activates/deactivates chosen bundles.

The configuration of the experiment is an XML file which describes three types of data:

- bundle groups (components) – a set of identifiers denoting version sets of OSGi bundles;
- repository – a folder whose each sub-folder (named according to one bundle group identifier) contains the corresponding bundle version set, i.e. the .jar files of individual versions of the given bundle;
- setting – a pre-defined combination of bundle versions which will be run on experiment startup, creating a particular configuration of the application.

An experiment is initiated by loading the configuration, upon which the initial set of bundle versions according to the default setting is installed and started on the underlying OSGi framework. The GUI then works as a special-purpose shell



```
<csdemo>
    <repository path="."></repository>
    <components>
        <component name="Server" />
        <component name="Client" />
        <component name="Hello" />
    </components>
    <settings>
        <setting name="default">
            <settingItem componentName="Server" jarName="MessagePrinterOsgiServer_v1.1.0.jar" />
            <settingItem componentName="Hello" jarName="Hello3_1.0.0.201104192054.jar" />
        </setting>
        <setting name="test1">
            <settingItem componentName="Server" jarName="MessagePrinterOsgiServer_v1.0.0.jar" />
            <settingItem componentName="Client" jarName="MessagePrinterOsgiClient_v1.0.0.jar" />
            <settingItem componentName="Hello" jarName="Hello1_1.0.0.201104192052.jar" />
        </setting>
        <setting name="test2">
            <settingItem componentName="Server" jarName="MessagePrinterOsgiServer_v1.1.0.jar" />
            <settingItem componentName="Client" jarName="MessagePrinterOsgiClient_v1.1.0.jar" />
            <settingItem componentName="Hello" jarName="Hello2_1.0.0.201104192054.jar" />
        </setting>
        <setting name="testInvalid">
            <settingItem componentName="Bubo" jarName="MessagePrinterOsgiServer_v1.1.0.jar" />
            <settingItem componentName="Hello" jarName="Hello3_1.0.0.201104192054.jar" />
        </setting>
    </settings>
</csdemo>
```

Figure 4: Tool configuration

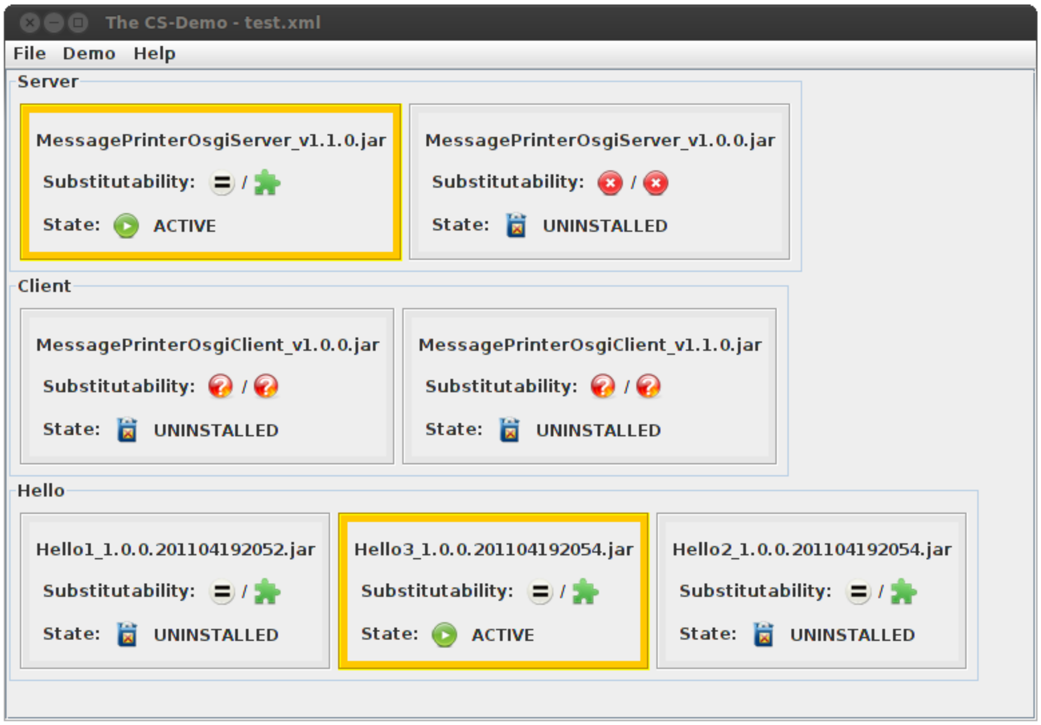
to the framework, enabling the user to change the configuration of the application in terms of its composing bundles selected from the available bundle groups.

Each time a bundle in the bundle versions group is activated, the substitutability of the other versions in the group against the (newly) running one are re-evaluated in the changed context of other running and installed bundles. The information about substitutability is indicated to the user by the icons on each bundle rectangle.

6 Conclusion

In this paper we presented the formal definition and practical implementation of a component substitutability verification method. Its key contributions are the novel use of the component's deployment context to enable safe substitution for non-subtype replacement components, and the ability to provide sufficiently strong formal guarantees of type consistency even when applied on current industrial component frameworks.

The current implementation of the substitutability verifier has several shortcomings. At the bundle representation and comparison level, it does not handle fragment bundles and optional imports. It also intentionally omits dynamic imports and bundle dependencies (known bad practices in the OSGi world).



Server

MessagePrinterOsgiServer_v1.1.0.jar
Substitutability: = / 🧩
State: ▶ ACTIVE

MessagePrinterOsgiServer_v1.0.0.jar
Substitutability: ✗ / ✗
State: UNINSTALLED

Client

MessagePrinterOsgiClient_v1.0.0.jar
Substitutability: ? / ?
State: UNINSTALLED

MessagePrinterOsgiClient_v1.1.0.jar
Substitutability: ? / ?
State: UNINSTALLED

Hello

Hello1_1.0.0.201104192052.jar
Substitutability: = / 🧩
State: UNINSTALLED

Hello3_1.0.0.201104192054.jar
Substitutability: = / 🧩
State: ▶ ACTIVE

Hello2_1.0.0.201104192054.jar
Substitutability: = / 🧩
State: UNINSTALLED

Figure 5: The GUI of the tool

Also, the chosen architecture — implementing the tool as user-space bundles — enables its portability but prevents integration into the bundle installation/resolving/updating process (which would be a desirable goal since it would provide user-transparent compatibility verification). These issues are the subject of further improvements of the implementation.

This type-based substitutability verification method can be wrapped into easy to use tools and data that promote its practical use. One such practical extension implemented by our team is the automated creation of correct semantic version identifiers for the OSGi framework. Another application, the contextual substitutability experiment demonstrator, has been described in the previous section.

Concerning further research, the formal definitions of the method should be extended to clusters of components (e.g. to support safe substitution of larger subsets of applications) and applied more specifically to inter-component relations in dynamic architectures. The practical implementation for OSGi will need to supply the missing aspects of the component model, and overcome the issues of tighter integration in the frameworks.

References

[1] Bauml, J. and P. Brada, *Automated versioning in OSGi: a mechanism for component software consistency guarantee*, in: *Proceedings of Euromicro SEAA* (2009).

[2] Bauml, J. and P. Brada, *Reconstruction of type information from java bytecode for component compatibility*, in: *5th workshop on Bytecode Semantics, Verification, Analysis and Transformation (Satellite Event of ETAPS 2010)*, Paphos, Cyprus, 2010 .

[3] Belguidoum, M. and F. Dagnat, *Formalization of component substitutability*, Electronic Notes on Theoretical Computer Science **215** (2008), pp. 75–92.

[4] Beugnard, A., J.-M. Jézéquel, N. Plouzeau and D. Watkins, *Making components contract aware*, Computer **32** (1999), pp. 38–45.

[5] Brada, P., "Specification-Based Component Substitutability and Revision Identification," Ph.D. thesis, Charles University in Prague (2003).

[6] Brada, P. and L. Valenta, *Practical verification of component substitutability using subtype relation*, in: *Proceedings of the 32nd Euromicro SEAA conference* (2006), pp. 38–45.

[7] Bruneton, E., T. Coupaye, M. Leclercq, V. Quéma and J.-B. Stefani, *The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems*, Software Practice and Experience **36** (2006), pp. 1257–1284.

[8] Cardelli, L., *Type systems*, in: *Handbook of Computer Science and Engineering*, CRC Press, 1997 .

[9] Chaki, S., E. Clarke, N. Sharygina and N. Sinha, *Verification of evolving software via component substitutability analysis*, Formal Methods in System Design **32** (2008).

[10] Cook, W. R., W. Hill and P. S. Canning, *Inheritance is not subtyping*, in: *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1990), pp. 125–135.

[11] Desnos, N., M. Huchard, C. Urtado, S. Vauttier and G. Tremblay, *Automated and unanticipated flexible component substitution*, in: *Proceedings of 10th International Symposium on Component-Based Software Engineering*, Lecture Notes in Computer Science **4608/2007** (2007).

[12] Georgas, J., A. van der Hoek and R. Taylor, *Using architectural models to manage and visualize runtime adaptation*, IEEE Computer **42** (2009), pp. 52–60.

[13] Gosling, J., B. Joy, G. Steele and G. Bracha, "The Java Language Specification, Third Edition," Prentice Hall, 2005.

[14] Jeek, K. and P. Brada, *Compatibility verification of components in terms of functional and extra-functional properties - tool support*, in: *Proceedings of the 12th International Conference on Enterprise Information Systems - Information Systems Analysis and Specification* (2010), pp. 510–514.

[15] Leavens, G., K. Leino and P. Mller, *Specification and verification challenges for sequential object-oriented programs*, Formal Aspects of Computing **19** (2007), pp. 159–189.

[16] Mach, M., F. Plášil and J. Kofron, *Behavior protocol verification: Fighting state explosion*, International Journal of Computer and Information Science **6** (2005), pp. 22–30.

[17] McCamant, S. and M. D. Ernst, *Formalizing lightweight verification of software component composition*, in: *Proceedings of SAVCBS 2004: Specification and Verification of Component-Based Systems*, Newport Beach, CA, USA), 2004, pp. 47–54.

[18] The OSGi Alliance, "OSGi Service Platform Core Specification," (2009), release 4, Version 4.2.

[19] Plášil, F. and S. Višnovský, *Behavior protocols for software components*, IEEE Transactions on Software Engineering **28** (2002).

[20] Polakovic, J., S. Mazare, J.-B. Stefani and P.-C. David, *Experience with safe dynamic reconfigurations in component-based embedded systems*, in: *Proceedings of 10th International Symposium on Component-Based Software Engineering*, Lecture Notes in Computer Science **4608/2007** (2007).

[21] Stuckenholz, A., *Component updates as a boolean optimization problem*, Electronic Notes on Theoretical Computer Science **182** (2007), pp. 187–200, proceedings of the Third International Workshop on Formal Aspects of Component Software (FACS'06).

[22] Szyperski, C., *Component technology - what, where, and how?*, in: *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, Portland, Oregon, 2003.

[23] Taylor, R. N., N. Medvidovic and P. Oreizy, *Architectural styles for runtime software adaptation*, in: *Proceedings of Joint Working IEEE/IFIP Conference on Software Architecture 2009 & European Conference on Software Architecture 2009*, 2009.

[24] Vallecillo, A., J. Hernández and J. M. Troya, *Component interoperability*, Technical Report ITI-2000-37, Universidad de Málaga, Spain (2000).

[25] Wegner, P. and S. B. Zdonik, *Inheritance as an incremental modification mechanism or what like is and isn't like*, , **322** (1988), pp. 55–77.