TYPED ABSTRACTIONS FOR CLIENT-SERVICE INTERACTIONS IN OSGI

Sven De Labey and Eric Steegmans
University of Leuven, Department of Computer Science, 200A, Celestijnenlaan, B-3000 Leuven, Belgium

Keywords: Service-Oriented Computing, OSGi, Object-Oriented Programming, Language Extension.

Abstract: The Open Services Gateway initiative (OSGi) is a successful attempt to bridge the gap between Java and Service Oriented Computing. OSGi provides an LDAP-based query language for fine-tuning service retrieval and offers an eventing mechanism that signals changes to a service's lifecyle to all clients depending on that service. Nonetheless, a number of challenges remain unsolved. OSGi's service query language, for instance, bypasses important compile-time guarantees on the syntactical correctness of queries and the language works only for properties that never change during the lifetime of a service. What programmers need, however, is a statically type-checked, robust query language that takes into account dynamically evolving, volatile service characteristics. A second problem is that the lifecycle management system requires programmers to write a considerable amount of boilerplate logic for reacting to service events. This obfuscates the business logic, which in turn decreases code comprehension and increases the odds for introducings bugs when implementing client-service interactions.

This paper evaluates OSGi as a platform for programming client-service interactions in Java. After focusing on a number of shortcomings of OSGi's integrated service query language and its lifecycle management system, we propose a solution based on a programming language extension. After the conceptual definition of these new language concepts, we show how they can be transformed to regular Java code without losing interoperability with the OSGi standard.

1 INTRODUCTION

Object-Oriented programming languages such as Java are increasingly adopting the paradigm of Service-Oriented Computing (Papazoglou, 2003). One of the most popular SOA adopters is the Open Services Gateway initiative (OSGi, 2006). OSGi technology provides a service-oriented, component-based environment and offers standardized ways to manage the software lifecycle (Marples and Kriens, 2001). It subscribes to the *publish-find-bind* model by providing a *central service registry* which is used by providers to publish their services along with relevant metadata. Such registered services can then be retrieved by clients by means of an LDAP-based search mechanism.

Next to providing functionality for dynamic service registration and retrieval, OSGi also supports *dynamic reconfiguration* of service architectures. Based on an extension of the Event Listener pattern, the OSGi middleware notifies service clients of important lifecycle changes of the services they depend on (OSGi, 2004). This is a key feature of OSGi because service architectures are inherently dynamic and volatile: services can be added, migrated, updated and removed, but at least, clients are given a chance to react to these events.

OSGI is a successful attempt to bridge the gap between Object-Oriented Programming and Service-Oriented Computing, but at the same time, it still imposes a lot of responsibilities on the programmer (Hall and Cervantes, 2004). Its integrated service query language, for instance, is very weak and bypasses compile-time guarantees on the syntactic correctness of a query. Also, the benefits of the lifecycle notification system are overshadowed by the requirement to write a considerable amount of boilerplate code, which bypasses compile-time guarantees in a similar way.

In this paper, we evaluate the OSGi middleware as a means for implementing client-service interactions in Java-based service-oriented architectures. We identify a number of problems and we show how these can be solved by ServiceJ, our Java extension that

De Labey S. and Steegmans E. (2008).
TYPED ABSTRACTIONS FOR CLIENT-SERVICE INTERACTIONS IN OSGI.
In *Proceedings of the Third International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 157-166
DOI: 10.5220/0001766701570166

(1) increases the level of abstraction and (2) provides compile-time guarantees on the correctness of service queries. Next, we show that code for dealing with lifecycle changes can be transparently injected during the compilation from ServiceJ to Java, as such freeing programmers from having to implement non-functional boilerplate code.

This paper is structured as follows. Section 2 provides a comprehensive review of OSGi in the context of client-service interactions. Based on problems defined in that section, Section 3 proposes a Java extension, ServiceJ, that aims at solving the problems of OSGi. The implementation of ServiceJ is described in Section 4. Related work is presented in Section 5 and we conclude in Section 6.

2 EVALUATION OF CLIENT-SERVICE INTERACTIONS IN OSGI

OSGi is a dynamic module system for Java with built-in support for dynamic reconfiguration of components (called "bundles" in OSGi). Within the context of this paper, the most interesting property of OSGi is that it follows the *publish-find-bind* methodology of Service-Oriented Computing. During the activation of a bundle in an OSGi environment, the bundle gets the opportunity (1) to *publish* its own services in the registry and (2) to *search* that registry for those services on which the bundle itself depends. OSGi extends this dependency management mechanism by including *an eventing mechanism* that notifies bundles of important changes to a service's lifecycle. An office component depending on a printer service, for instance, can create an event listener that notifies the office component whenever a printer service is registered, modified, or unregistered.

This section provides a comprehensive review of service interactions in OSGi. First, we explain and evaluate how services can be registered along with their metadata (Section 2.1). Then, we describe what actions clients need to undertake before they can consume these services (Section 2.2). Next, we evaluate how clients are confronted with lifecycle changes that occur at the services they depend on (Section 2.3). A solution to the problems identified in this section is proposed in further sections of this text.

2.1 Service Registration

A bundle uses the central service registry to register the services it offers to other bundles. This registry is accessed through a bundle's *bundle context*, which is configured by the OSGi runtime system on bundle activation. During registration, the *service object* is added to the registry, along with the relevant metadata describing characteristics of that service. This metadata is represented as a *dictionary* containing key-value pairs.

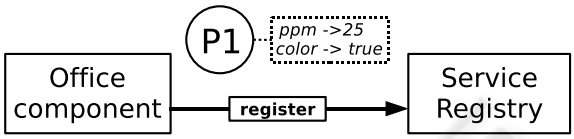

Figure 1: Service Registration in OSGi.

Figure 1 shows how an OfficeComponent registers a PrinterService (named P1) along with a dictionary describing that the printer's paper output expressed in pages per minute equals 25 (`ppm -> 25`) and that it provides color printing (`color -> true`). Listing 1 shows how this registration process can be programmed in OSGi by interacting with the bundle's BundleContext(represented by context).

```
1  //--1-- Specify metadata Properties
2    metadata = new Properties();
3    metadata.put("ppm","25");
4    metadata.put("color","true");
5  //--2-- Register Service and Metadata
6  ServiceRegistration registration =
7    context.registerService(
8      PrinterServiceImpl.class.getName(),
9      printer,
10     metadata);
```

Listing 1: Registering services and metadata in OSGi.

Evaluation. Support for metadata is a major strength of OSGi as it enables potential clients to fine-tune their service selection strategy based on service-specific characteristics. One major drawback, however, is that this query mechanism is poorly integrated with Java, as there are no *static guarantees* on the syntactical correctness of the metadata. OSGi accepts wrongly typed key-value pairs such as `(ppm,-2)` or `(ppm,true)` even though these pairs do not make sense. The Java compiler is unable to detect these problems because metadata are added as strings.

A second problem is that clients must *know* the names of the properties that were added during registration (such as "`ppm`") as well as the domain of possible values for each property, but there is no standardized way to retrieve this information.

2.2 Service Retrieval

Most bundles are providers and consumers at the same time. Next to registering services, then, they

```
1  //--1-- Retrieving Services in OSGi
2  public PrinterService searchPrinterService(){
3     ServiceReference[] printerReferences;
4     try {
5       String serviceType = "(objectClass="+PrinterService.class.getName()+")";
6       String serviceFilter="(&"+serviceType+"(&(ppm>=25)(color=true)))";
7       references[] = context.getServiceReferences(null,serviceFilter);
8       context.addServiceListener(this, serviceFilter);
9       return (PrinterService) context.getService(references[0]);
10    }
11    catch(InvalidSyntaxException ise){
12      return null; //exception is thrown when the query string contains errors
13    }
14 }
15 //--2-- Reacting to Service Lifecycle Changes in OSGi
16 public void serviceChanged(ServiceEvent event){
17    switch (event.getType()) {
18       case ServiceEvent.REGISTERED:
19          this.printer = (FirstService)context.getService(event.getServiceReference());
20          break;
21       case ServiceEvent.MODIFIED:
22          this.printer = (FirstService) context.getService(event.getServiceReference());
23          break;
24       case ServiceEvent.UNREGISTERING:
25          //...remove reference and try to find another one
26          break;
27    }
28 }
```

Listing 2: Retrieving OSGi services (top) and reacting to changes to their lifecycle (bottom).

also need to *find* services in order to successfully execute their business operations. Service retrieval is carried out by sending *LDAP-based queries* to the central OSGi service registry. A typical *service query* contains two important pieces of information. First, the *service type* is used to specify what kind of service is requested. The service registry needs this information to search for *conforming* services. A query like objectClass=PrinterService.class, for instance, triggers the service registry to return references to those services that specified PrinterService as one of their values for the objectClass property (i.e. the LDAP property representing the service type). Second, a *filter* can be defined to further fine-tune service retrieval. These filters relate to the metadata that was attached to a service during registration, as discussed in Section 2.1.

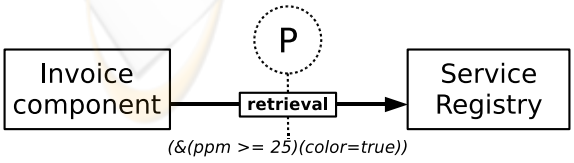

Figure 2: Service retrieval in OSGi.

Figure 2 shows how a color printer (depicted as P) is requested that prints at least 25 pages per minute. The variables used in the LDAP query (ppm and color) refer to metadata entered by the service provider. The code required for carrying out such a service retrieval is shown in the top part of Listing 2 (lines 1–13). First an LDAP-like query is specified, constraining the service type (line 5) with the abovementioned Quality of Service requirements (line 6). Then, the query is executed using the bundle's BundleContext (line 7), and a service object is returned (line 9). An InvalidSyntaxException may occur during this process (line 10–12) when the query contains syntax violations or typing errors.

Evaluation. To evaluate the expressive power of this query mechanism, we consider three kinds of *service properties* in which service clients may be interested: (1) *static*, (2) *dynamic* and (3) *derived* properties:

- *Static Service Properties.* OSGi's LDAP-based query mechanism is ideally suited for static properties, as these properties never change. Metadata such as *pages per minute* can be registered along with a PrinterService instance because this information is assumed not to change during the lifetime of the printer.
- *Dynamic Service Properties.* Properties that *do* change when a service is operational, introduce major problems. The queue of a PrinterService,

for instance, grows and shrinks as jobs arrive and get processed. Obviously, these properties cannot be added during registration, so OSGi's metadata system does not support them. Consequently, clients cannot impose constraints on dynamic service properties when searching for suitable services.

- *Derived Service Properties.* A third class of characteristics comprises information that depends on input provided by the client bundle. The cost for printing a file, for instance, may be calculated by combining the file's page count with the candidate service's cost for printing one page. But the OSGi metadata system does not support this. Consequently, in their LDAP service queries, clients cannot specify constraints on information that is *derived* from the metadata of a service.

  A related shortcoming is that service selection cannot be *optimized* based on service-specific characteristics. It is impossible, for instance, to select the PrinterService with the *minimal* cost for printing a given file. Indeed, *minimizing costs* is a *function* and the application of functions or quality metrics to a set of candidate services is currently not supported by OSGi.

Next to shortcomings relating to dynamic and derived properties, OSGi also lacks the benefits of a *statically typed*, *comprehensible* query language (as shown on lines 5–6 in Listing 2). Similar to the service registration process where syntactically incorrect metadata could be added, it is possible to write inconsistent or syntactically incorrect queries. These errors will only be detected at runtime when the query is parsed. Programmers using LDAP-based queries must therefore expect to catch an InvalidSyntaxException *every time* they want to retrieve services (cfr lines 11–13 in Listing 2). This is in sharp contrast with regular method invocations, of which the Java compiler provides strong guarantees about their syntactical correctness and the abscence of typing errors.

In summary, the OSGi query mechanism is a *dynamically* typed query language that only deals with *static* service characteristics. What we need, however, is a *statically* typed query language that is able to deal with *dynamic, volatile* service properties.

2.3 Service Lifecycle Management

The primary challenge in OSGi environments is the handling of inter-bundle dependencies (OSGi, 2004). Such dependencies occur when the successful execution of a business method in one bundle depends on the existence and proper working of one or more services published by *other* bundles. In this situation, it is paramount that the depending bundle has a means for *tracking* the lifecycle of those external services. In OSGi, an event-notification mechanism referred to as the Whiteboard pattern (OSGi, 2004) is integrated so as to allow service clients (i.e. depending bundles) to track the lifecycle of those services on which they depend.

A bundle uses its BundleContext to add *listeners* that notify the bundle of any changes to a service's lifecycle. To fine-tune this notification mechanism, the bundle may attach an LDAP-based *event filter* to this listener, similar to the queries that were used for service retrieval. In that case, only changes to the lifecycle of services satisfying the LDAP filter will be signaled. An example is shown in Listing 2 on line 8. This operation causes the bundle to be notified whenever a service satisfying the serviceFilter constraint is registered, modified, or unregistered. The handling of these events are programmed in the serviceChanged method (lines 16–29).

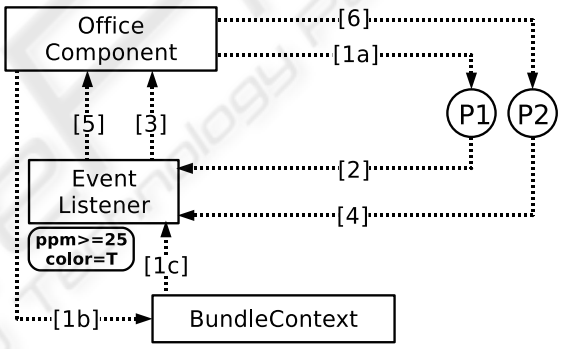

Figure 3: Lifecycle changes to OSGi services.

Figure 3 shows a typical reaction to a lifecycle change. On bundle activation, the OfficeComponent searches for a PrinterService **(1a)** and asks its BundleContext to create an EventListener along with an *LDAP event filter* **(1b–1c)** such that the OfficeComponent is informed of any changes to the lifecycle of the PrinterService. This service is unregistered unexpectedly, causing an *unregistration event* to be sent to all the interested event listeners **(2)**, one of which eventually delivers it to the OfficeComponent **(3)**. The latter now removes all references to this service. After a while, a second PrinterService, P2, joins the SOA. Assuming that this new service satisfies the LDAP filter of the event listener, the OfficeComponent now receives a notification through its EventListener **(4–5)**. It reacts to this event by binding its `printer` variable to P2, the new PrinterService **(6)**.

Evaluation. The first problem with this approach is related to the limited expressive power of LDAP-

based *event filters*. These filters cannot prevent the client bundle from receiving *spurious events*. An example of such a spurious event is when multiple `PrinterService` instances satisfy the event filter that was attached to the event listener. In that case, all changes to the lifecycle of these services will be signaled, *even when the bundle does not depend on all of these services*. In other words, the bundle is notified about events occurring at totally unrelated services that happen to satisfy the bundle's LDAP event filter. Bundles are thus forced to check whether the event source is identical to the service they depend on *before* reacting to an event notification; and failure to do so (such as our sample in Listing 2) may lead to erroneous runtime behaviour that is extremely hard to debug.

A second disadvantage is the lack of separation of concerns between business logic and technical, non-functional code. OSGi forces programmers to write a considerable amount of boilerplate code for building and registering event handlers, and this must be done for *every* service a bundle depends on. Ideally, technical middleware interactions for handling service registration, modification, and unregistration are hidden for programmers, since dealing with these non-functional concerns would firmly increase the complexity of implementing the business logic. In OSGi, however, these event notifications are mixed with the business logic, and it is the responsibility of the programmer to react on any problem that may occur when the client bundle interacts with an external service.

Another problem that remains unsolved is what should be done when a client bundle engages in *multiple, consecutive* interactions with a remote service. Clients receiving a service unregistration event in the middle of such a client-service transaction, cannot transparently *restart* their transaction by switching to an alternative service endpoint. Rather, the unregistration event would trigger a search for an alternative service, and the transaction would then continue as if nothing happened. In other words, the first part of the transaction would be executed on the service being unregistered, whereas the second part would be executed on another, unrelated service. This lack of support for bundling related client-service interactions into a client-service transaction thus violates important properties such as *atomicity* and *isolation*.

Design Goals

In this paper, we focus on the realization of three important design goals concerning the definition of a programming language suitable for implementing client-service interactions in volatile service architectures:

- **Typed Abstractions.** Programmers must have the same compile-time guarantees about the correctness of service interactions as they have when programming local object interactions. Among others, this creates a need for a statically typed query language.
- **Expressiveness.** Service architectures are dynamic and volatile by nature. Therefore, a service query language must be able to take into account *dynamic* and *derived* service properties.
- **Transparency.** The distributed nature of services living in loosely coupled, unrelated bundles makes it impossible for clients to be sure of the *availability* of a service. It is paramount that such volatility and availability problems are hidden for programmers as much as possible, since failure to do so obfuscates the business logic of the service-oriented application. Changes to a service's lifecycle, as well as general availability problems and other technical middleware issues must be transparently dealt with by the programming model and by its accompanying middleware.

In the next Section, we propose an extension to the Java programming model that integrates specialized support for implementing client-service interactions. We show how these language concepts can be used for the implementation of OSGi applications, and we compare them to the original OSGi concepts.

3 TYPED ABSTRACTIONS FOR OSGI

In this section, we show how the language concepts introduced by our Java extension (called ServiceJ) can be used as a statically checkable service query language. We also show how the transformation from ServiceJ to Java allows for the transparent injection of code for dealing with lifecyle modification events sent by external services. The overall goal of this extension is to exonerate programmers from having to deal with the technical issues introduced by Service-Oriented Computing, and to provide them with static guarantees on the correctness of their client-service interactions. Section 3.1 shows how OSGi service retrieval is improved by our extension. Then, Section 3.2 shows how lifecycle changes such as service modification and service unregistration are transparently handled. Finally, Section 3.3 shows how ServiceJ eases service registration in OSGi service architectures.

3.1 Finding and Binding Services

ServiceJ introduces (1) *type qualifiers* to identify variables depending on services, and (2) *declarative operations* for fine-tuning the set of candidate services that can be assigned to a variable that depends on a service. Both extensions are discussed below.

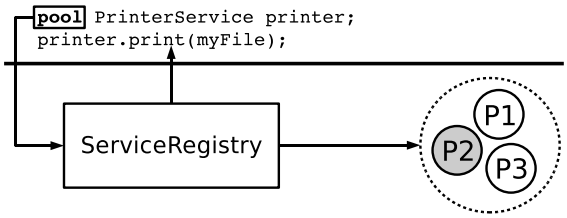

Figure 4: The pool qualifier triggers transparent service lookup based on the service type (PrinterService), and supports transparent service binding and service failover.

Type Qualifiers. ServiceJ introduces type qualifiers to distinguish variables holding service references from variables pointing to local objects. This differentiation allows the ServiceJ-to-Java transformer to inject additional operations for transparently dealing with the typical challenges introduced by service architectures. Currently, two type qualifiers are defined:

- *The* `pool` *Qualifier*. This type qualifier is used to indicate that a variable depends on a service published by another bundle. A declaration such as "pool PrinterService printer" indicates that the bundle depends on a PrinterService exported by some other bundle. The advantage of signaling this information, is that the ServiceJ-to-Java transformer can *inject* special middleware interactions for initializing the variable. This exonerates programmers from interacting with the OSGi middleware so as to configure all their service dependencies. The `pool` qualifier causes the transformer (see Section 4 for details) to transparently inject operations for (1) service retrieval, (2) non-deterministic service binding, and (3) service failover. Because it installs a non-deterministic service selection protocol, the `pool` qualifier is typically used when all type-conforming services in the architecture are assumed to be *interchangeable*.
- *The* sequence *Qualifier*. Sometimes, certain services are preferred above others based on service-specific characteristics. In that case, a *deterministic* service selection procedure is required. The `sequence` qualifier, which is a *subqualifier* of the pool qualifier, is used to decorate variables depending on external services that require a deterministic selection strategy driven by *preferences* that were specified by means of the declarative orderby operation, which is explained below.

Example. Figure 4 depicts how a reference to a `PrinterService` is decorated with the `pool` qualifier. It shows how the programmer is exonerated from implementing interactions with the OSGi middleware in order to obtain service references. Indeed, programmers simply declare their service variable with a `pool` qualifier and start invoking operations without initializing it. Initialization is now the responsibility of the ServiceJ middleware, which selects an appropriate service (P2) before invoking the `print` operation. Should P2 fail during this interaction, then ServiceJ automatically injects another service into the `pool` variable and reinvokes the operation.

Declarative Operations. Similar to the LDAP-based query language provided by OSGi, ServiceJ incorporates specialized support for fine-tuning service selection. In contrast with OSGi, however, these operations are now fully integrated within the programming languages in the form of *declarative operations*. In ServiceJ, queries no longer refer to untyped metadata, but instead, they directly relate to the operations that are exported by the service's interface. Instead of using an untyped property such as "`ppm`", for instance, queries in ServiceJ refer to a *public inspector method* such as getPagesPerMinute(), which is exported by the PrinterService interface. This provides better compile-time guarantees on the syntactical and conceptual correctness of queries. Currently, two declarative operations are defined for fine-tuning service selection:

- *The* `where` *Operation*. This operation is used to *constrain* a set of candidate services according to a number of business requirements and Quality of Service constraints. Thus, the `where` operation can be used to replace OSGi's untyped query language. The service query from Listing 2, for instance, can be translated using the `where` operation as follows:

```
pool PrinterService p
            where p.getPPM()>=25 &&
                  p.supportsColor();
```

  This enables the compiler to provide more guarantees on the correctness of the query when compared to the string-based LDAP queries of OSGi. But the `where` operation is also more expressive than basic LDAP expressions. Indeed, by using operations published in the service interface, our query language can take into account the most up-to-date information of a service, thus allowing programmers to impose constraints on *dynamic*

and *derived* service properties, whereas OSGi's current query language can only take into account static metadata information that was entered during service registration. An additional benefit is that the `where` operation is combined with the `pool` qualifier, implying that these constrained sets of candidate services still provide transparent service selection, injection and fail-over.

- *The* `orderby` *Operation.* This operation is used to sort a set of candidate services according to the preferences of a user. In the PrinterService example, a programmer can use the orderby operation to select the printer that minimizes the cost for printing a given file. Such a query can be implemented as follows:

```
sequence PrinterService p
          orderby p.getCostFor(file);
p.print(myFile);
```

Note that this query necessitates the use of the `sequence` qualifier because a *deterministic* service selection policy is requested. Detailed information about the use of type qualifiers and their associated service selection strategies can be found in our previous paper (De Labey, S. and Steegmans, 2007).

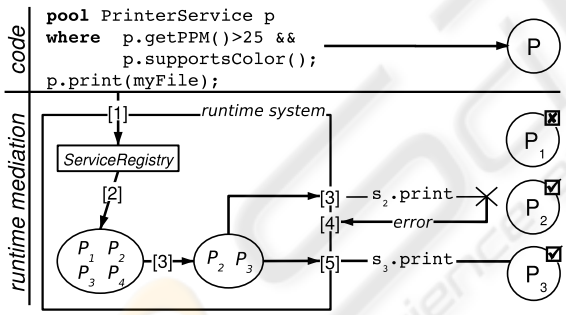

Figure 5: Runtime view of constrained service set.

Example. Figure 5 shows how declarative operations are combined with type qualifiers so as to create a higher level of abstraction for the programmers of client-service interactions. At the level of the source code, no references to physical services, nor any other interactions with the OSGi middleware are found. The programmer only specifies the service type (PrinterService) along with the relevant constraints that must be satisfied by the service (as an argument of the `where` operation). This is enough to invoke the `print` operation in Figure 5.

The middleware, then, uses this information to query the OSGi service registry **(1)** to obtain a set of candidate services **(2)**. This set is filtered, retaining only those services that satisfy the constraint that was specified by the user in a `where` clause. From this constrained set, a service is non-deterministically selected, and the `print` operation is invoked on it **(3)**. This unexpectedly returns an error **(4)**, triggering the ServiceJ middleware to transparently select another services from the constrained pool **(5)** before reinvoking the operation on that new service endpoint.

3.2 Transparent Lifecycle Management

In Section 2.3, we have shown that lifecycle management is a tedious task: programmers have to write code for the registration of event listeners that yield spurious notifications due to the limited expressiveness of OSGi's LDAP-based event filtering expressions. In ServiceJ, on the other hand, all events relating to non-functional concerns such as service registration, modification and unregistration are handled *transparently* by the middleware. Given a service interaction such as printer.print(myFile) in Figure 5, the ServiceJ middleware transparently searches for a service reference and injects it into the printer variable. Notifications about changes to this service's lifecycle are handled by the middleware. Service unregistration, for instance, is handled by releasing the service reference, and by looking for an alternative service that satisfies the constraints that where imposed on the `pool` by means of `where` or orderby clauses.

Service Sessions. One remaining challenge is the execution of a complex business operation comprising *multiple interactions* between the client bundle and an external service. For consistency reasons, it is crucial that the pool variable is bound to the *same service* for the entire operation. This calls for a *coarser-grained*, transactional failover mechanism that extends our basic failover strategy (as explained in Section 3.1). ServiceJ supports these operations by introducing **session**$\{\ldots\}$ blocks to combine related service interactions into *sessions* (De Labey, S. and Steegmans, 2007). On entering a session block, the runtime injects a service into the pool variable as described in Section 3.1. From then on, the pool variable is *locked* until the session ends successfully, or until the injected service becomes unreachable, e.g. due to service unregistration. In the latter case, the runtime *aborts* the current session, *unlocks* the pool variable, injects another pool member, and then restarts the entire session. By introducing these session-scoped locks on pool variables, ServiceJ allows developers to think of a session as an *atomic, fault-tolerant interaction* with an external service.

Example. Figure 6 shows an example of a *service session* with a PrinterService. First, the file to be printed is uploaded to the printer server, which returns an `id` identifying the print job. Then, the client does some extra processing, and finally it invokes the `print` operation with the `id` as an argument. To illustrate the runtime behaviour of a client-service transaction encapsulated in a session block, we assume that the PrinterService unregisters between these two client-service interactions.

The runtime system first invokes the `upload` operation on `P1` **(1)**, which returns and `id` and then unregisters **(2)**. Unregistration is signaled to the client by means of an *unregistration event.* This event is transparently handled by the runtime system, which injects a new service into the pool variable (`p`) and then restarts the entire service session. The `upload` operation is invoked on this new service, which returns a new `id` **(4)**, and eventually, the `print` operation is invoked on `P2` with the correct `id` **(5)**.

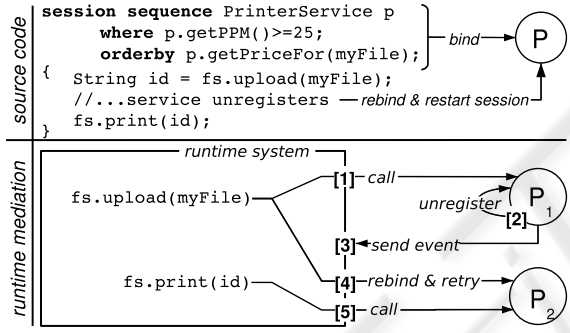

Figure 6: Service sessions provide a basic transaction mechanism for complex client-service interactions.

3.3 Publishing Services

The main objective of the ServiceJ programming model is to provide an appropriate programming language for implementing client-service interactions, but this programming model is also beneficial for *service providers* since service registration is simplified as well. In stead of registering untyped metadata in dictionaries that bypass important compile-time guarantees, programmers may now implement inspector methods to describe service characteristics. Rather than adding key-value pairs like `(ppm,25)`, for instance, programmers now implement operations such as getPagesPerMinute() that are exported by the service interface. The use of these inspector methods not only allows service queries to take into account *dynamic* and *derived* properties (see Section 3.1), but it also solves the problem of managing *changing characteristics* of services. Indeed, no unregistrations or renewed registrations are required to update the characteristics of a service object, since the metadata is fully integrated inside that service object. This is different in OSGi, where an external dictionary contains the metadata information. Changes to a service object, then, must be reflected in the dictionary, and this often requires unregistration and renewed registration, which both produce a large number of events.

4 IMPLEMENTATION

We have specified a number of goals concerning the *implementation* of these new language concepts. First, it is paramount that the resulting program is *OSGi-compliant*. This is because OSGi provides support for other technical challenges outside the domain of ServiceJ, such as installing, resolving and activating bundles, as well as wiring bundles together and managing these compositions. Second, we want to reuse the standard Java compiler and the Java Virtual Machine, which requires ServiceJ applications to be translated to OSGi-compliant Java applications as a *preprocessing step*. This section focuses on how this preprocessing step is realized.

Pre-transformation. Figure 7 shows how ServiceJ code **(1)** is read by a lexer and a parser so as to create a *ServiceJ metamodel instance*. During this process, the semantically poor nodes of the abstract syntax tree that was created by the ServiceJ parser are transformed to semantically rich instances of metamodel classes **(2)**. This metamodel instance, then, is fed to the ServiceJ-to-Java transformer, which is responsible for transforming the ServiceJ metamodel instance into an equivalent Java metamodel instance **(3)**.

Transformation. The input of the transformation is a ServiceJ metamodel instance representing the ServiceJ application. This metamodel is an extension of Jnome (van Dooren, M. et al., 2007), our Java metamodel. It introduces classes modeling the newly introduced language concepts. Pool variables, for instance, are modeled as a MemberVariable with an instance of the Pool metamodel class attached to it. During the transformation of this MemberVariable, the transformer will detect the presence of this qualifier, and it will inject into the Java metamodel all necessary OSGi interactions for initializing the pool variable. Also, when a method is invoked on a pool variable the transformer will detect that the target of the method invocation refers to an instance of the Pool metamodel class. Figure 7 shows what actions are undertaken in this situation: the transformer injects spe-

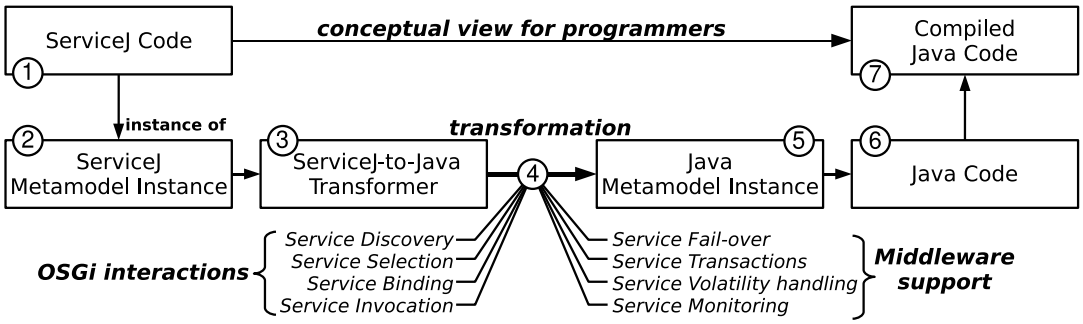

Figure 7: Transformation of ServiceJ code to compiled Java code.

cific OSGi interactions, such as *service discovery* and *service selection*, and also makes the code more robust by injecting middleware support for *service failover* and *service transactions* **(4)**.

Post-transformation. After the equivalent Java metamodel has been built by the ServiceJ-to-Java transformer, a simple code writer **(5)** transforms the Java metamodel instance to *.java* files **(6)**. These files can be compiled by the standard Java compiler, thus finishing the compilation process **(7)**.

While the compilation of ServiceJ programs is an extensive process, it is important to note that all this is entirely shielded for developers. The transformation procedure is best seen as a black box consuming ServiceJ source files and producing compiled Java classes. Developer intervention is never required during this process.

5 RELATED WORK

Our language concepts most closely resemble the constructs that are introduced in object-oriented languages such as WebOz (Hadim and et al., 2000), WebL (Kistler and Marais, 1998), and XL (Florescu et al., 2003) to enable service failover. But those concepts rely on hardwired service references, so they fail to capture the *volatility* of services in an OSGi architecture, and they cannot support dynamic service discovery. These languages also lack support for defining constraints and selection policies, whereas our extension integrates that support by means of declarative operations.

Jini (Sun, 2005) is a close competitor to OSGi, since it also attempts to bring Service-Oriented Computing to the world of object-oriented programming (Huang and Walker, 2003). One drawback of Jini with respect to OSGi, is that Jini's query mechanism is much weaker than OSGi's LDAP-based queries. Jini relies on *entry objects* representing the *exact* value that a property must have, whereas OSGi allows programmers to use operators such as "<" and ">=" to specify *ranges* of values, rather than a single value. Moreover, Jini does not install an event notification architecture, but instead relies on a *leasing* system instead. Clients retrieving a Jini service are given a lease representing the time they are allowed to use the service. Leases must be renewed temporarily, which also introduces a programming overhead.

Cervantes and Hall observed that OSGi does not provide any support for managing service dependencies apart from the basic event notification system in (Hall and Cervantes, 2003) and (Cervantes and Hall, 2003). They propose to improve dependency management based on *instance descriptors*. Such instance descriptors are XML files describing how a bundle depends on external services. A `<requires>` tag is introduced to specify the *service type*, the *cardinality* of the dependency, and the *filter condition.* Programmers may also define the *bind* and *unbind* methods that should be called when a service is to be bound or unbound. One problem with this approach is that the information to be specified is *untyped*, even though it directly refers to public methods provided by the bundle. Also, the filter condition is now isolated in an XML file, but it is still an LDAP-based query string, which lacks support for dynamic and derived service properties. Moreover, instance descriptors create a strong dependency between a Java file and an XML file, and they divide important decisions concerning the business logic between these two files.

Our approach of introducing type qualifiers and qualifier inference is based on the approach followed in Javari (Tschantz and Ernst, 2005). We have proven the type soundness of this language extension in a way similar to the Java extension presented in (Pratikakis et al., 2004). For more information about the formal development of ServiceJ, we refer to (De Labey, S. et al., 2006) and (De Labey, S. and Steegmans, 2007).

6 CONLUSIONS & FUTURE WORK

The Open Services Gateway Initiative is a successful attempt to bridge the gap between object-oriented programming and service-oriented computing, but a number of challenges remain unsolved. In this paper, we have focused on problems stemming from the limited expressiveness, comprehensibility and static guarantees that the OSGi's LDAP-based query language provides. We have also shown that the lifecycle management system requires too much programmer intervention and that it too often signals spurious events.

To solve these problems, we propose an integration of ServiceJ language concepts into the OSGi programming model. First, *type qualifiers* allow the ServiceJ-to-Java transformer to inject additional instructions for transparently handling service failures and lifecycle changes. Second, *declarative operations* allow programmers to fine-tune service selection in a type-safe way. Third, programmers can demarcate client-service transactions using *session blocks*, leaving the complex management of such transactions entirely to the ServiceJ middleware.

Future Work. A proper event-notification system should also support the notification of *functional events*. These are events that directly relate to the business logic of an application (e.g. an event signaling that a file is successfully printed). We plan to extend ServiceJ's programming model so as to integrate language support for working with this second class of events.

REFERENCES

Cervantes, H. and Hall, R. (2003). Automating Service Dependency Management in a Service-Oriented Component Model. In *Proceedings of the 6th Workshop on Foundations of Software Engineering and Component Based Software Engineering*, pages 379–382.

De Labey, S. and Steegmans, E. (2007). ServiceJ. A Type System Extension for Programming Web Service Interactions. In *Proceedings of the Fifth International Conference on Web Services (ICWS07)*.

De Labey, S., van Dooren, M., and Steegmans, E. (2006). ServiceJ: Service-Oriented Programming in Java. Technical Report KULeuven, CW451, June 2006.

Florescu, D., Gruenhagen, A., and Kossmann, D. (2003). XL: A Platform for Web Services. In *Proceedings of the First Conference on Innovative Data Systems Research*.

Hadim, M. and et al. (2000). Service Combinators for Web Computing in Distributed Oz. In *Conference on Parallel and Distributed Processing Techniques and Appl.*

Hall, R. and Cervantes, H. (2003). Gravity: supporting dynamically available services in client-side applications. *SIGSOFT Software Engineering Notes*, 28(5):379–382.

Hall, R. and Cervantes, H. (2004). Challenges in building service-oriented applications for OSGi. *Communications Magazine, IEEE*, 42(5):144–149.

Huang, Y. and Walker, D. (2003). Extensions to Web Service Techniques for Integrating Jini into a Service-Oriented Architecture for the Grid. In *Proceedings of the International Conference on Computational Science*.

Kistler, T. and Marais, H. (1998). WebL - A Programming Language for the Web. In *7th Intl. Conference on the World Wide Web*.

Marples, D. and Kriens, P. (2001). The open service gateway initiative: An introductory overview. *IEEE Communications Magazine*, 39(12).

OSGi (2004). Listeners considered harmful: The whiteboard pattern. In *www.osgi.org/documents/osgi_technology/*.

OSGi (2006). *Open Services Gateway Initiative Specification v4.0.1 – http://www.osgi.org*.

Papazoglou, M. (2003). Service Oriented Computing: Concepts, Characteristics and Directions. In *Proceedings of the 4th International Conference on Web Information Systems Engineering*.

Pratikakis, P., Spacco, J., and Hicks, M. (2004). Transparent Proxies for Java Futures. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 206–223, New York, NY, USA. ACM.

Sun (2005). *The Jini Architecture Specification and API Archive – http://www.jini.org*.

Tschantz, M. S. and Ernst, M. D. (2005). Javari: Adding reference immutability to Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, pages 211–230, San Diego, CA, USA.

van Dooren, M., Vanderkimpen, K., and De Labey, S. (2007). The Jnome and Chameleon Metamodels for OOP.