Towards a Framework for Behavioral Specifications of OSGi Components

Jan Olaf Blech

fortiss GmbH, Munich, Germany

We present work on behavioral specifications of OSGi components. Our behavioral specifications are based on finite automata like formalisms. Behavioral specifications can be used to find appropriate components to interact with, detect incompatibilities between communication protocols of components and potential problems resulting from the interplay of non-deterministic component specifications. These operations can be carried out during development and at runtime of a system. Furthermore, we describe work carried out using the Eclipse based implementation of our framework.

1 Introduction

Traditional software component systems come – if at all – with a basic typing that indicates possible values at the component interface. In our work, we are extending this view to specify possible behavior of components in addition to the basic typing. We use a typing that encapsulates protocols specified by finite automata based descriptions. We present a first version of an implementation for the OSGi [19] framework. OSGi allows dynamic reconfiguration of Java based software systems. We demonstrate a tool based approach that allows the specification of method call based communication protocols, and the formalization of creation and deletion of components during a system's lifetime. We check possible behavior of interacting components for behavioral compatibility including deadlocks. We can resolve possible incompatibilities by choosing options from non-deterministic behavioral specifications and – after discovery of a potential incompatibility – reacting inside the components accordingly.

This work describes efforts towards an operationalization and an implementation of a behavioral types framework for OSGi. It realizes parts of our vision described in [6]. Unlike our work presented in [7], it is realized entirely using Java technology and is aimed towards the OSGi component system. A more comprehensive version of our OSGi semantics is described in a report [5]. In this paper, we primarily address protocol based behavior of components. The new contributions of this work comprise:

- A formal definition of the OSGi semantics that is suitable for the abstract view that our behavioral types provide on OSGi.

- A first implementation of a finite automata based behavioral type system for OSGi that integrates different tools and workflows into a framework.

- Early versions of editors and related code for supporting adaption and checking.

- An exemplarily integration of behavioral type checkers comprising minimization, normalization and comparison. One checker has been implemented in plain Java. Additionally we have integrated a checker and synthesis tool presented in [12] for deciding compatibility, deadlock freedom and detecting conflicts in non-deterministic specifications at runtime and development time.

- Usage scenarios (interaction protocols) of our behavioral types for OSGi at runtime and development time.

- The modeling of an example system: a booking system to show different usage scenarios.
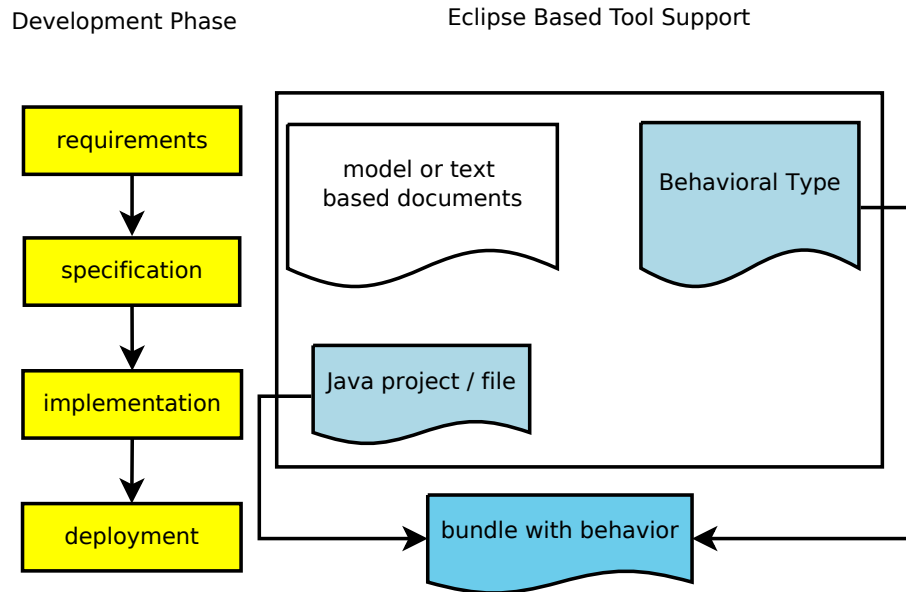
Figure 1: Behavioral types at development time

1.1   Our Setting

Figure 1 shows the development chain supported by our framework. The development of our Eclipse based development process for OSGi bundles can roughly be divided into four phases. Bundles are top-level OSGi components that aggregate classes, Java packages and deployment information. The four phases are supported by our behavioral descriptions in the following way.

- In the requirements and specification phase, after the component / bundle structure has been determined, one can start using our tools for behavioral types. Requirements on components and specified protocols for component interaction can be described by using the automata like specification mechanisms provided by our behavioral types.

- During the implementation, one creates bundles which contain the OSGi bundle information: static dependencies, classically typed interface descriptions, objects to be created at start of the bundle and Eclipse specific plugin information, e.g., extensions to the user interface. In addition to this we add our behavioral descriptions. These are given as files and can become accessible through the OSGi registration service. The OSGi registration service keeps tracks of objects / services provided by bundles and their properties.

- At deployment and runtime of the system one has bundles including their behavioral specifications. These are 1) registered at the OSGi infrastructure and 2) can be used to discover appropriate components. Components can further use these (as shown in Figure 2) to decide 3) whether and how they want to interact, to discover potential incompatibilities and ways to resolve them. Decision may be based on algorithms and tools which are provided as separate bundles.

1.2   Related Work

Interface automata [1] are one form of behavioral types. Like in this work, component descriptions are based on automata. The focus is on communication protocols between components which is one aspect

1) start of a bundle   2) discover components

bundle with behavior   bundle with behavior

register
behavioral   ask for
descriptions   specification

OSGi infrastructure

bundle with behavior   compatibility
                                                  checker
get behavior   (bundle)

bundle with behavior
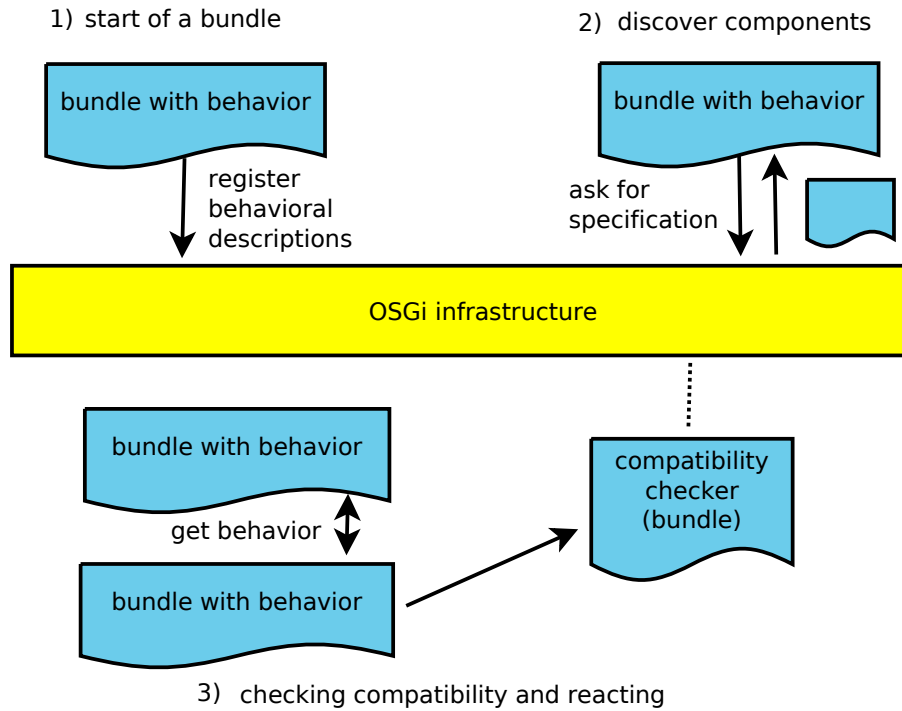
3) checking compatibility and reacting

Figure 2: Behavioral types at runtime

that we also address in this paper. Interface automata are especially aimed at compatibility checks of different components interacting at compile time of a system. Behavioral types have also been used in the Ptolemy framework [18] with a focus on real-time systems.

Specification and contract languages for component based systems have been studied in the context of web services. A process algebra like language and deductive techniques are studied in [9]. Another process algebra based contract language for web services is studied in [8]. Emphasize in the formalism is put on compliance, a correctness guaranty for properties like deadlock and livelock freedom. Another algebraic approach to service composition is featured in [14].

JML [11] provides assertions, pre- and postconditions for Java programs. It can be used to specify aspects of behavior for Java methods. A similar description mechanism has been used for systems specified in synchronous dataflow languages like Lustre [13]. Assertion like behavioral specifications have also been studied in the context of access permissions [10].

Behavioral types as means for behavioral checks at runtime for component based systems have been investigated in [3]. In this work, the focus is rather put on the definition of a suitable formal representation to express types and investigate their methodical application in the context of a model-based development process.

A language for behavioral specification of components, in particular of object oriented systems – but not OSGi –, is introduced in [16]. Compared to the requirement-based descriptions proposed in our paper, the specifications used in [16] are still relatively close to an implementation. Recent work regarding refinement of automata based specifications is, e.g., studied in [20].

To the best of our knowledge, existing work does describe OSGi and its semantics only at a very high level. Other behavioral type like frameworks do not exist for OSGi up till now. A specification of the OSGi semantics based on process algebras is featured in [22]. Some investigations on the relation
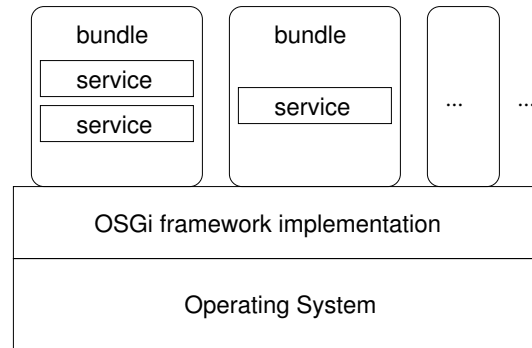
Figure 3: OSGi framework

between OSGi and some more formal component models have been done in [17]. Means for ensuring OSGi compatibility of bundles realized by using an advanced versioning system for OSGi bundles based on their type information is studied in [4]. Aspects on formal security models for OSGi have been studied in [15].

1.3   Overview

We present OSGi and our formalization of its semantics in Section 2. Section 3 introduces our automata based behavioral types specification mechanism. Operations on behavioral types at development and at runtime are described in Section 4 for the OSGi framework. Implementation of the framework using Eclipse / OSGi techniques is described in Section 5. Section 6 exemplifies the use of behavioral types and its operations for a booking system and a conclusion is given in Section 7.

2   OSGi and its Semantics

We present an overview on OSGi following our description in [6] and present a formalization of the semantics based on our more detailed report [5].

The OSGi framework is a component and service platform for Java. It allows the aggregation of Java packages and classes into bundles (cf. Figure 3) and comes with additional deployment information. The deployment information triggers the registration of services for the OSGi framework. Bundles provide means for dynamically configuring services, their dependencies and usages. OSGi bundles are used as the basis for Eclipse plugins but also for embedded applications including solutions for the automotive domain, home automation and industrial automation. Bundles can be installed and uninstalled during the runtime. For example, they can be replaced by newer versions. Hence, possible interactions between bundles can in general not be determined statically.

Bundles are deployed as .jar files containing extra OSGi information. This extra information is stored in a special file inside the .jar file. Bundles generally contain a class implementing an OSGi interface that contains code for managing the bundle, e.g., code that is executed upon activation and stopping of the bundle. Upon activation, a bundle can register its services to the OSGi framework and make it available for use by other bundles. Services are implemented in Java. The bundle may itself start to use existing services. Services can be found using dictionary-like mechanisms provided by the OSGi framework. Typically one can search for a service which is provided using an object with a specified Java interface.

In the context of this paper, we use the term OSGi component as a subordinate concept for bundles, objects and services provided by bundles.

The OSGi standard only specifies the framework including the syntactical format specifying what bundles should contain. Different implementations exist for different application domains like Equinox[1] for Eclipse, Apache Felix[2] or Knopflerfish[3]. If bundles do not depend on implementation specific features, OSGi bundles can run on different implementations of the OSGi framework.

A Method-Call Semantics

In the following we provide a formal semantics for OSGi. We concentrate on capturing behavior originating from method calls between different bundles and objects. Memory and exchange of data between these bundles and objects is not taken into account. Thus, we provide an overapproximation – in the sense of possible behavior – and abstraction of a real system.

Object and method definitions An object is defined as a tuple $(m_0, ..., m_n)$ comprising constructor and method definitions $m_0, ..., m_n$. Since we incorporate constructors into this tuple it cannot be empty.

The semantics of an object is given by the semantic interpretation of its methods and its object state. The semantics of a method is giving by an automaton (L, E, l_0) comprising a set of locations L an initial location $l_0 \in L$ and edges $E = (l_i, M, l_j)$ between locations. An addition to source and target location l_i and l_j an edge comprises a set (can be ordered) of method calls and special calls M. These can be tuples $(m, o, b) \in M$ comprising a method definition m of an object o that is associated with a bundle b. Furthermore, M can contain special calls for: adding and removing bundles, and creating and deleting objects. Each transition from E represents an action that is atomic or non-terminating to the method but not to the OSGi system. It can represent a memory update, but also other method calls. A method call can itself trigger a non-terminating method in the same or in other objects. Therefore a transition does not necessarily represent a terminating operation.

An object state is a set of tuples $\{(m_n, l_{n_i}, id_n, cs_n), ..., (m_p, l_{p_j}, id_p, cs_p)\}$ comprising active method status states $(m_n, l_{n_i}, id_n, cs_n), ..., (m_p, l_{p_j}, id_p, cs_p)$. Each tuple represents a method call, consisting of a method definition, its actual locations, an identifier id and a call state cs.

The call state is part of an active method status state. It is a set of method definitions and method id plus status information for which the active method is waiting to return. The id is used to distinguish different calls to the same method.

Bundles From an operational semantics point of view, bundles aggregate objects into units that are enumerated in the OSGi system and can be loaded and removed during runtime by user commands or from other bundles. A bundle is a set of objects $\{o_{activator}, ...\}$ comprising an object $o_{activator} = (..., m_{start}, m_{stop}, ...)$ which is created on activation. It comprises two distinguished methods m_{start}, m_{stop} which are called during activation and deactivation.

In an implemented OSGi system, the $o_{activator}$ object has to implement the `BundleActivator` interface defined in `org.osgi.framework`. It comprises two methods with signatures:

```
void start(BundleContext context) throws java.lang.Exception
```

---

[1]`http://www.eclipse.org/equinox/`
[2]`http://felix.apache.org/site/index.html`
[3]`http://www.knopflerfish.org/`

and

```
void stop(BundleContext context) throws java.lang.Exception
```

The semantical definition of bundle states aggregates its object states. A bundle state is defined as a set of object states $\{s_{o_i}, ..., s_{o_k}\}$ for object states $s_{o_i}, ..., s_{o_k}$. Like for bundles, objects, and methods, we distinguish between a system state and a system definition – capturing a systems architecture. Both can change during the lifetime of a system. A standard OSGi system has one (as, e.g., in the Equinox framework implementation) or more bundles which are active at startup.

OSGi systems and OSGi system states   An OSGi system is a set of bundles. It comprises a distinguished bundle b_{init} which is activated at start-up. Analog to object and bundle state, we define an OSGi system state. A system state is defined as a set of bundle states $\{s_{b_i}, ..., s_{b_k}\}$ for bundle states $s_{b_i}, ..., s_{b_k}$. The initial state of an OSGi system comprises the start of the *start* method in the activator object of the initial bundle. The initial state of an OSGi system is defined as $s_{init} = \{s_{b_{init}}\}$ with $s_{b_{init}} = \{o_{activator}\}$ and $o_{activator} = \{(m_{start}, l_{start_0}, 0, \emptyset)\}$.

Dynamic architecture of OSGi systems   An important aspect of our formalization is the impact on OSGi operations that can change the structure of OSGi systems. Such operations can be triggered by OSGi methods themselves, e.g., comprising adding and removing objects and bundles. Another option is to perform these operations by a command line interface (e.g., starting Eclipse with the console option using Equinox) at runtime on the OSGi framework. Here, we distinguish the following structure changing operations on OSGi systems: Starting / loading a system, adding a bundle and activating it, removing a bundle (and deactivating it), adapting a bundle and its services, closing / removing a system. Characteristic for these operations is the fact that new behavior becomes possible or is removed at runtime of the OSGi system. Thus, the semantics of an OSGi system and possible events can in general not be determined statically at the start of a system.

State transitions in OSGi   State transitions can modify both, structure of a system and the state of objects, bundles and a system. They are made up from local transitions appearing within methods and from handling terminated methods. In general state transitions are highly non-deterministic and define a relation of

$$\text{previous system state} \times \text{previous system definition} \times$$
$$\text{next system state} \times \text{next system definition}$$

For an OSGi system $S = \{..., b, ...\}$: We regard the system state $s = \{..., s_b, ...\}$ with $s_o \in s_b$ and $(m, l_i, id, cs) \in s_o$. From here, the following basic state transition cases can be distinguished:

- Calling a method m' of object o' from a bundle b': We regard a transition $(l_i, M, l_j) \in o$ with $o \in b$. The following steps are performed.
    1. The step can be performed under the preconditions that $(m', o', b') \in M$ and o' and b' exist in S.
    2. cs is updated by adding the method call indicating its bundle, object and id.
    3. A new element $(m', l'_0, id', \emptyset)$ is added to the object state where m' belongs to o'. id' is a new identifier for the method m'.

- Executing a method step: We regard a transition $(l_i, M, l_j) \in o$ with $o \in b$.

1. The step can be performed under the precondition that $cs = \emptyset$.
2. s_o is updated as $s'_o = s_o/(m, l_i, id, cs) \cup \{(m, l_j, id, cshandle(M))\}$. Thus, (m, l_i, id, cs) is removed and $(m, l_j, id, cshandle(M))$ is added instead.
   cshandle transforms M into a representation that indicates which methods have been called and keeps track of their ids. Furthermore, *cshandle* takes care of special operations that modify the system definition.

- Returning from a method call: Any method status state with $cs = \emptyset$ and no edge that may lead to a possible succeeding state can be processed in the following way:
  1. The method status state is removed.
  2. The call state of any method that m has called is updated such that the entry for the m call is removed.

Furthermore, the following operations are handled:

- Adding a bundle : The cs from any object state s_o with $(m, l_i, id, cs) \in s_o$ can contain a special operation (denoted: add bundle b') for adding a bundle b' and changing the system definition from S into $S' = S \cup \{b'\}$.

- Removing a bundle: The cs from any object state s_o with $(m, l_i, id, cs) \in s_o$ can contain a special operation for removing a bundle b' (denoted: remove bundle b') and changing the system definition from S into $S' = S/\{b'\}$.

- Creating an object. The cs from any object state s_o with $(m, l_i, id, cs) \in s_o$ can contain a special operation (denoted: create object (o', b)) for adding an object o' and changing a bundle definition $b \in S$ to $b' = b \cup \{o'\}$. The system definition is, thus, changed from S into $S' = S/b \cup \{b'\}$.

- Deleting an object: The cs from any object state s_o with $(m, l_i, id, cs) \in s_o$ can contain a special operation (denoted: delete object (o', b)) for deleting an object o' and changing a bundle definition $b \in S$ to $b' = b/o'$. The system definition is, thus, changed from S into $S' = S/b \cup \{b'\}$.

Key characteristics of the OSGi semantics   The method call semantics described above features some key-characteristics of OSGi:

- State transitions in bundles and objects are triggered out of the bundles and objects themselves. They only involve the component where they originate from and components that are interacted with during a state transition. The rest of the system remains untouched with respect to the underlying abstractions of our semantics.

- Different components run asynchronously as long as there is no method call between them.

- Method calls provide synchronization points between components.

- Method calls are blocking.

3   Behavioral Types as Specification Mechanism

Our framework essentially supports finite automata for specifying expected incoming, potential outgoing method calls, the creation and deletion of components during a time span and other events that may occur in the lifetime of a system. A component's behavior can be specified by one or multiple automata each one describing a behavioral aspect. Formally, we have an alphabet of labels Σ, a set of locations L, an initial location l_0 and a set of transition edges E where each transition is a tuple (l, σ, l') with $l, l' \in L$ and $\sigma \in \Sigma$. These are aggregated into a tuple to form a behavioral specification:
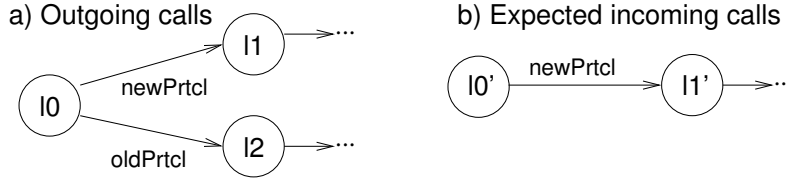
Figure 4: Supporting different protocol versions

$$(\Sigma, L, l_0, E)$$

This view abstracts from the specifications given in Section 2. Our intention is to define interaction protocols or some aspects of them like the expected order of incoming and outgoing method calls for a component. Specifications for different components are independent of each other as long as there is no method call (e.g., indicated by the same label name) in the specifications.

Example: Two components interacting Specifications can be used for different behavioral aspects. Figure 4 shows two excerpts of automata for outgoing and expected method calls from two different component specifications:

$$(\{newPrtcl, oldPrtcl, ...\}, \{l0, l1, l2, ...\}, l0, \{(l0, newPrtcl, l1), (l0, oldPrtcl, l2), ...\})$$

and

$$(\{newPrtcl, ...\}, \{l0, l1, ...\}, l0, \{(l0, newPrtcl, l1), ...\})$$

Here, the first component can do two different method calls in its initial state: newPrtcl, oldPrtcl. The second component expects one method call newPrtcl in its initial state. In this case both components may interact with each other, if both components use the newPrtcl.

4   Checking Compatibility and Making Components Compatible

We describe operations that can be used at development and at runtime of a system. The operations use behavioral types from Section 3. Furthermore, we briefly describe the handling of potential incompatibilities discovered by type comparison at runtime within a software system.

4.1   Simple Behavioral Type Checking

We have developed and implemented different operations for handling and comparing behavioral types, for deciding compatibility and for deadlock freedom.

Simple comparison for equality of types and comparison for refinement between two automata based specifications involves the following steps.

- A basis for the comparison of two types is the establishment of a set of semantical artifacts (e.g., method calls) that shall be considered. The default is to use the union of all semantical artifacts that are used in the two types. Comparison for refinement is achieved by eliminating certain semantical artifacts from this set. For consistency this also requires eliminating associated transitions from the types or, depending on the desired semantics, replacing an edge with an empty or τ label.

- It is convenient to complete specifications for further comparison: Specification writer may only have specified method calls or other semantical artifacts that trigger a state change. Here, we

automatically add an error location. We collect possible labels and for locations that do not have an edge for a label leading to another location indicating a possible semantical artifact, we add edges with the missing label to the error location.

- In case of specifications which have been completed and that have no locations with two outgoing edges with the same labels, we perform a minimization of automata based specifications. This way, we merge locations and get rid of unnecessary complexity automatically.

- Normalization of automata based specifications. This, involves the ordering of edges and in some cases locations with respect to the lexicographic order of their labels / location names.

- Checking for equality involves the checking of equality of the labels on edges. Optionally, one can also consider the equality of location names of an automaton. Location names may imply some semantics but in our standard settings they only serve as ids. When location names serve only as ids, we construct a mapping between location names of the two automata involved in the comparison operation.

These operations have been implemented in Java. They do not need additional tools or non-standard plugins.

4.2 Deciding Compatibility and Deadlock-Freedom

In addition to the operations described in Section 4.1 we have adapted a SAT and game-based tool – VissBIP presented in [12] – to serve as a compatibility and deadlock checker for our behavioral types for OSGi. Our framework uses VissBIP to support the checking of the following properties:

- Deadlocks checking: deadlocks resulting from potential sequences of method calls can be detected.

- Compatibility: A component anticipating a certain behavior of incoming method calls matches potential behavior of outgoing method calls by other components.

VissBIP uses a simplified version of the BIP semantics [21]. A system comprises concurrent automata with labeled edges. The automata synchronize with each other by performing edges with the same labels in parallel. Otherwise, the default case is that automata do not synchronize with each other. For comparing method call based behavioral specifications we use VissBIP on specifications that comprise expected incoming and outgoing method calls of components. In OSGi synchronization between components happens only when one component calls a method of the other component as indicated in the behavioral specification and the OSGi semantics. On the VissBIP side this corresponds to same labels in the automata that represent the behavior. In addition to the label compatibility checking, VissBIP is able to perform the introduction of priorities.

4.3 Runtime Adaption of Systems

One way of runtime adaption is the reaction to potential deadlocks or incompatibilities. Recall Figure 4: it shows behavioral specifications of two components which intend to communicate with each other. Possible outgoing method calls of one component and expected incoming method calls of the other component are shown. It can be seen that the first component is able to communicate using two different protocols: one starts by calling an initialization method newPrtcl, the other one starts by calling an initialization method oldPrtcl. The other component expects the newPrtcl call.

When we give these two specifications to VissBIP, it will return a list of priorities where the new-Prtcl edge is favored over the oldPrtcl edge in the first specification. In a Java implementation the first component can use this to dynamically decide at runtime which protocol to use.

- First, the component loads its own behavioral specification and the specification of the expected method calls of the second component. Technically, we support loading files and the registration of models as properties / attributes of bundles as provided by the OSGi framework.

- Next, we invoke VissBIP or another checking routine. Passing the behavioral specifications as parameters.

- The checking routine gives us a list of priorities. In the Java code we have a switch statement as a starting point for handling the different protocols. We check the priorities and go to the case for the appropriate protocol.

Thus, in addition to deadlock detection, we can use behavioral specifications for coping with different versions of components and desired interacting protocols.

4.4   Component Discovery at Runtime

A central feature of our behavioral descriptions for OSGi components is registering them to a central OSGi instance. In order to inform other components of the existence of a bundle with behavioral offers and needs, we register its behavioral properties using the OSGi service registry belonging to a `BundleContext` which is accessible for all bundles in the OSGi system:

```
registerService(java.lang.String[] clazzes,
    java.lang.Object service,
    java.util.Dictionary<java.lang.String,?> properties)
```

Here, we register a collection of behavioral objects as properties for a service representing a bundle under a String based key. In our framework, we register a collection of behavioral models as "BEHAVIOR". The behavioral models are loaded from XML files that are integrated into the bundle. The behavioral models come with meta information which identify the parts of the behavior of a bundle which they describe. The service itself is represented as an object. Additional interface information is passed using the `clazzes` argument.

5   Tool Support during Development and at Runtime

The features described in this paper have been implemented in Eclipse. Our framework offers the following ingredients and is build using the following concepts:

- EMF/.ecore based meta model of behavioral descriptions for easy interactions with other Eclipse based tools. Each specification is associated with a description which classifies what is actually specified, e.g., incoming method calls, outgoing method calls, component creation and deletion or something mixed.

- Editors for behavioral descriptions. Figure 5 shows a screenshot of an editor for automata based specifications.

- Other operations like abstractions, minimization and comparison of behavioral types (some of them are described in Section 4.1) are implemented. They can be used by referencing one of our plugins and can be extended.

- An integration of the VissBIP checker as Eclipse plugin / OSGi bundle and transformations for using it with our behavioral types are offered.
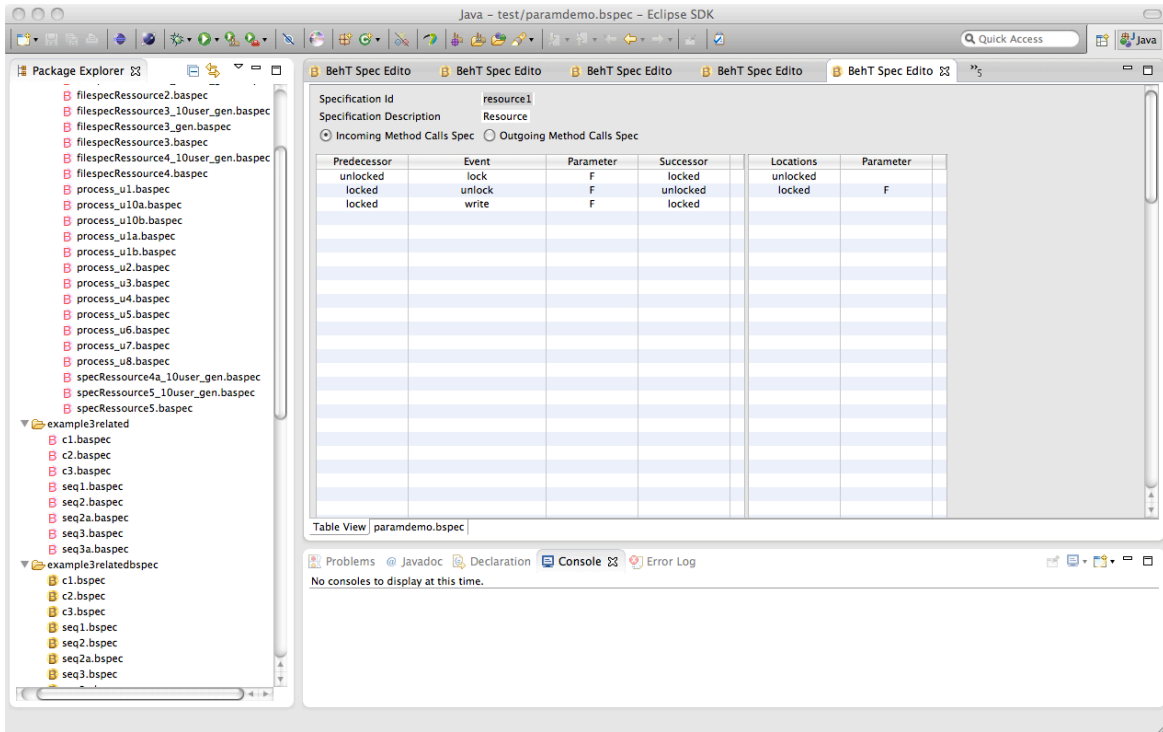
Figure 5: Specifying behavior using our editors in Eclipse

While the editors are only invoked at development time, the specifications and operations on them are used both: at development time and at runtime of the system for reconfiguration. At development time, they are invoked using the Eclipse front-end. At runtime, they are invoked using method calls to plugins that realize the operations and are deployed as OSGi bundles.

6 Behavioral Types for a Booking System

We present the use of behavioral types to highlight some features and usages of our work on an example: a flight booking system.

Figure 6 shows the main ingredients of our flight booking system. Clients are served by middleware processes which are created and managed by a coordination process. Middleware processes use concurrently a flight database and a payment system. The described system is an example inspired by realistic systems where the middleware is implemented using Java/OSGi. In addition to the middleware components we describe databases and parts of the frontend using our behavioral types to make checks of these parts possible.

The following means of behavioral interaction can be distinguished:

- **Component calls between methods / communication protocol** In our flight booking system, a client can call a coordination process and middleware processes. Middleware processes can call
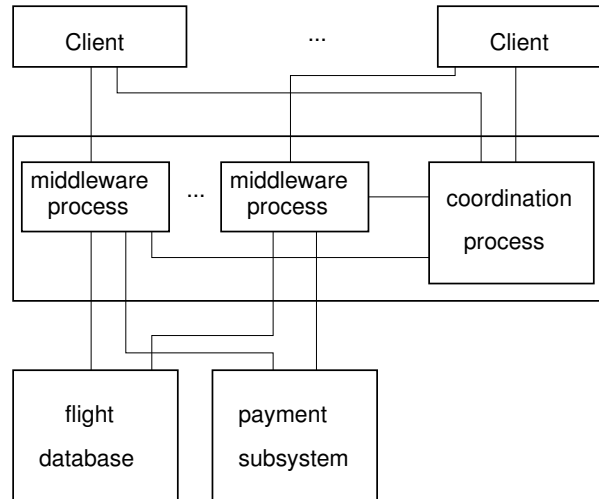
Figure 6: Components of our flight booking system

methods providing access to the flight database and the payment subsystem. The method calls
need to respect a distinct protocol which can be encoded using our behavioral types.

- **Creation and deletion of new components** The coordination process creates and removes mid-
  dleware process such that there is one process per client. Providing support for analysis of such
  dynamic aspects is a long term goal for our behavioral types but not in the scope of this work.

- **Concurrent access to shared resources** Middleware processes perform reservations, cancella-
  tions, rebookings, seat reservations and related operations on the flight database. These operations
  do require the locking of parts of the data while an operation is performed. For example, during a
  seat reservation a certain amount of the available seats in an aircraft is locked so that a customer
  can chose one without having to fear that another customer will chose the same seat at the very
  same time. In the current state we are able to provide some behavioral types support here.

Example: Specification of outgoing method calls of a middleware process   Specifications of pos-
sible expected incoming and potential outgoing method calls give information about a communication
protocol that is to be preserved. Typically different interaction sequences are possible, especially since
we are dealing with abstractions of behavior. In the booking system, a middleware process commu-
nicates with a flightdatabase (db) and the payment system (pay). The expected order of method calls
for a flight booking to these systems is shown in Figure 7. The figure shows only an excerpt of the
possible states and transitions. In addition to this, the initial state allows the start of a seat reservation
process and a cancellation process. Moreover, Figure 7 shows only the state changing method calls of
the behavioral specification of the booking process. Our real behavioral specification completely lists all
possible method calls in each state. This way, we can further analyze compatibility issues for example
with database systems that do not support all possible method calls of a middleware process.

In comparison to the outgoing method calls of a middleware process, the incoming method call spec-
ification is much simpler: A constructor call is performed by the coordination process upon initialization.
After that, the communication with the client is done using a webserver interface – comprising method
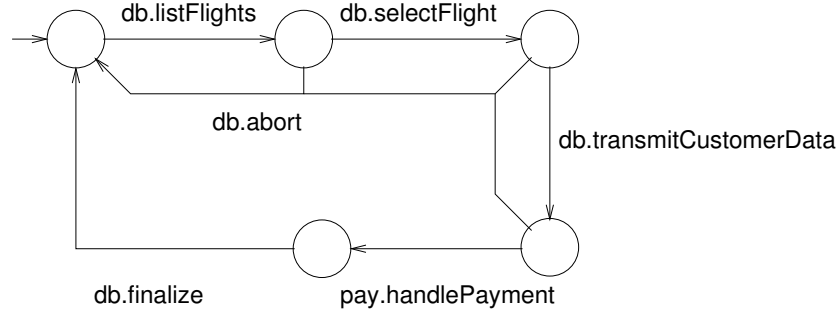calls that send raw request data to the middleware process and return raw response data that trigger,

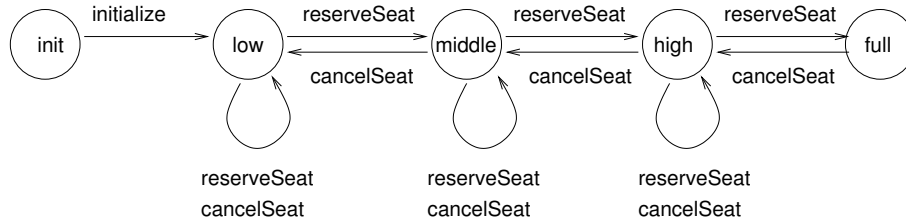Figure 7: Outgoing method calls of a middleware process

Figure 8: Behavioral model for seat reservation of a flight

e.g., displaying selected flights by the client – where no states in the communication process can be distinguished.

Example: Specification of database elements   Access to our database is done using method calls to a database process and is formalized using our automata based specification formalisms. The method calls result in locking and unlocking database elements. Seat reservation in a flight requires that a certain partition of the available seats is blocked during the selection process so that a client can make a choice.

Figure 8 shows our behavioral model of seat reservation for a single flight. Different loads are distinguished: low means that many seats are still available, while high means that only a few seats are available. The full state indicates that no additional seat reservations can be made, only cancellations are possible. The model is an abstraction of the reality since instead of treating each seat – potentially hundreds of available seats – independently we only distinguish their partitioning into four equivalence classes: low, medium, high and full.

Example: Database elements and deadlocks   Access to the flight database can result in deadlocks. The model from Figure 8 can serve as a basis for deadlock analysis. Consider the scenario shown in Figure 9: For each flight a different instance of the seat reservation model exists. Given three airports A, B and C: Suppose two people – person 1 and person 2 – want to fly from A to C via B. Seats for two flights need to be reserved: from A to B and from B to C. It is not desirable to reserve a seat from B to C if no seat is available for the flight to A to B. Otherwise, it might not be desirable to fly from A to B if no seat is available for the flight from B to C.

During the seat reservation a deadlock may occur: If person 1 reserves the last seat for the A to B flight before doing reservations for the B to C flight and person 2 reserves the last seat for the B to C flight before a seat reservation for the A to B flight a deadlock may occur, which may result in the cancellation of both journeys although one person could have taken the journey.

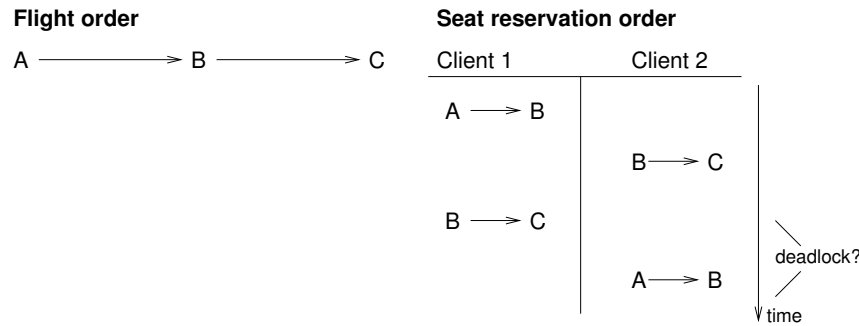**Flight order**   **Seat reservation order**

Figure 9: Concurrent seat reservation on two flights

If it is known before to the seat reservation system that person 1 and person 2 will fly from A to C – which is a reasonable assumption given the fact that they have entered their desired start and end destination into the system – we are able to detect such deadlocks. They can occur if both behavioral models of the seat reservation system are already in the high state – given that no other participants are doing reservations at this time we may also take compensating actions.

Evaluation   Modeling of the flight booking system has been carried out in several versions with several degrees of detail in our behavioral types framework plugins. Behavioral models are described as independent files. We have used our implemented operations on these files. Compatibility and deadlock checking can be performed without problems for several components interacting together. For our compatibility checking, we do not use all specifications of the entire system together but pick those that are relevant for a certain communication aspect.

7   Conclusion

We presented a first version of a framework for behavioral types for OSGi systems. In this paper, the main focus is on the OSGi semantics, the specification of behavior and checking the compatibility of specifications. Handling and reacting to specifications at runtime is another topic. We have described our implementation and its architecture.

So far, we are concentrating on Eclipse / OSGi systems. Other application areas for the future comprise 1) work towards behavioral types for distributed software services 2) work towards real-time embedded systems. This might require leaving the Java / OSGi setting, since these applications typically involve C code which communicates directly with – if at all – an operating system. There is, however, work on extensions for real-time applications of OSGi using real-time Java (e.g., [2]). Additional specification formalisms and the integration of new checking techniques are another challenge.

References

[1] L. de Alfaro, T.A. Henzinger. Interface automata. Symposium on Foundations of Software Engineering, ACM , 2001. doi:10.1145/503271.503226

[2] J. C. Américo, W. Rudametkin, and D. Donsez. Managing the dynamism of the OSGi Service Platform in real-time Java applications. Proceedings of the 27th Annual ACM Symposium on Applied Computing, ACM, 2012. doi:10.1145/2245276.2231952

[3] F. Arbab. Abstract Behavior Types: A Foundation Model for Components and Their Composition. Formal Methods for Components and Objects. vol. 2852 of LNCS, Springer-Verlag, 2003. doi:10.1007/978-3-540-39656-7_2

[4] J. Bauml and P. Brada. Automated Versioning in OSGi: A Mechanism for Component Software Consistency Guarantee. 35th Euromicro Conference on Software Engineering and Advanced Applications, 2009. doi:10.1109/SEAA.2009.80

[5] J. O. Blech. Towards a Formalization of the OSGi Component Framework. http://arxiv.org/abs/1208.2563v1. arXiv.org 2012.

[6] J. O. Blech, Y. Falcone, H. Rueß, Bernhard Schätz Behavioral Specification based Runtime Monitors for OSGi Services. Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), 2012. doi:10.1007/978-3-642-34026-0_30

[7] J. O. Blech, B. Schätz. Towards a formal foundation of behavioral types for UML state-machines. In: Proceedings of the 5th International Workshop UML and Formal Methods, 2012. doi:10.1145/2237796.2237814

[8] M. Bravetti, G. Zavattaro. A theory of contracts for strong service compliance. Mathematical Structures in Computer Science 19(3): 601-638, 2009. doi:10.1017/S0960129509007658

[9] G. Castagna, N. Gesbert, L. Padovani. A theory of contracts for Web services. ACM Trans. Program. Lang. Syst. 31(5), 2009. doi:10.1145/1538917.1538920

[10] N. Cataño and I Ahmed. Lightweight Verification of a Multi-Task Threaded Server: A Case Study With The Plural Tool. Proceeding of Formal Methods for Industrial Critical Systems (FMICS), vol 6959 of LNCS, Springer, 2011. doi:10.1007/978-3-642-24431-5_3

[11] P. Chalin, J.R. Kiniry, G.T. Leavens, E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. Formal Methods for Components and Objects, FMCO, vol. 4111 of LNCS, Springer 2005. doi:10.1007/11804192_16

[12] C. Cheng, H. Rueß, A. Knoll, C. Buckl. Synthesis of fault-tolerant embedded systems using games: from theory to practice. Verification, Model Checking, and Abstract Interpretation, vol. 6538 of LNCS, Springer 2011. doi:10.1007/978-3-642-18275-4_10

[13] J.-L. Colaço and M. Pouzet. Clocks as first class abstract types. EMSOFT, vol. 2855 of LNCS, Springer, 2003. doi:10.1007/978-3-540-45212-6_10

[14] J. L. Fiadeiro, A. Lopes. Consistency of Service Composition. Fundamental Approaches to Software Engineering (FASE), vol. 7212 of LNCS, Springer, 2012. doi:10.1007/978-3-642-28872-2_5

[15] O. Gadyatskaya, F. Massacci, A. Philippov. Security-by-Contract for the OSGi Platform. Information Security and Privacy Conference, SEC, Springer, 2012. doi:10.1007/978-3-642-30436-1_30

[16] E. B. Johnsen and R. Hähnle and J. Schäfer and Rudolf Schlatte and Martin Steffen. ABS: A Core Language for Abstract Behavioral Specification. Post Conf. Proceedings 9th Intl. Symposium on Formal Methods for Components and Objects 2010. Springer-Verlag 2010. doi:10.1007/978-3-642-25271-6_8

[17] M. Mueller, M. Balz, M. Goedicke. Representing Formal Component Models in OSGi. Proc. of Software Engineering, Paderborn, Germany, 2010.

[18] E.A. Lee, Y. Xiong. A behavioral type system and its application in ptolemy ii. Formal Aspects of Computing, 2004. doi:10.1007/s00165-004-0043-8

[19] OSGi Alliance. OSGi service platform core specification (2011) Version 4.3.

[20] C. Prehofer. Behavioral refinement and compatibility of statechart extensions. Formal Engineering approaches to Software Components and Architectures. Electronic Notes in Theoretical Computer Science, 2012.

[21] J. Sifakis. A framework for component-based construction – Extended Abstract. Software Engineering and Formal Methods, IEEE Computer Society, 2005. doi:10.1109/SEFM.2005.3

[22] H.A.M. Tchinda, N. Stouls, J. Ponge. Spécification et substitution de services osgi. Technical report, Inria (2011) http://hal.inria.fr/inria-00619233.