Ensuring OSGi Component Based Properties at Runtime with Behavioral Types

Jan Olaf Blech

RMIT University, Melbourne fortiss GmbH, Munich

Abstract. We present work on using automata based behavioral descriptions (behavioral types) of OSGi components for monitoring their specified behavior at runtime. Behavioral types are associated with OSGi components. We are focusing on behavioral types that specify protocols defined by possible orders of method calls of and between components and specifications based on the maximal execution time of these method calls. Behavioral runtime monitors for detecting deviations from a specified behavior are generated for components automatically out of their behavioral types. We sketch the integration of our behavioral runtime monitors into a behavioral types framework and present implementation and evaluation work on the behavioral runtime monitoring part.

1 Introduction

In our work, we are extending the basic typing concepts of traditional software component systems with means for specifying possible behavior of components. As with traditional types, like primitive datatypes and their composition, our *behavioral types* can be used for eliminating possible sources of errors at development time of software systems. This is analog to classical static type checks performed by a compiler. Furthermore, we can use behavioral types for eliminating possible sources of errors at runtime. This is analog to dynamic type checks performed when accessing pointers that reference data with types that can not be statically determined in some classical programming languages. Behavioral types also provide additional information about components which can be used for further tool based operations. Ensuring behavioral type correctness at runtime of an OSGi system is the main focus of this paper.

In this paper, we are using finite automata based descriptions of method call orders and maximal execution times of methods. Programmers even outside the academic community seem to be familiar with finite automata and thus, we believe that it is a good candidate for the acceptance of our specification formalism. We present a first version of an implementation [1] for the OSGi [14] framework [2]. OSGi allows dynamic reconfiguration of Java based software systems. In this paper, we concentrate on checking / ensuring of behavioral type safety at runtime of a system using *behavioral runtime monitors* generated from our behavioral types. We monitor a system's execution and throw behavioral types exceptions in case of deviations from the specified behavior.

[1] The parts of our behavioral types framework concerning behavioral runtime monitors as described in this paper are available at http://sourceforge.net/p/beht/wiki/Home

[2] Among other aspects of the framework, additional implementation details are described in [7]

Our work features the following highlights: 1) The use of multi-purpose automata based behavioral types. The same specification files can be used for other operations at compile time, e.g., static analysis of component compatibility, and runtime, e.g., discovery of components in a SOA like scenario, dynamic adaptation of components [6]. 2) The enforcement of these types at runtime by generating behavioral runtime monitors out of the types, using aspect oriented programming and an integration into Java by throwing runtime exceptions in case of deviation, and ensuring of maximal execution time of methods by using aspect oriented programming and runtime exceptions. There is no need to modify or add special comments to the source code files of the system. We think that this is highly beneficial for the acceptance of our approach since existing practices in Eclipse can still be used, people who are not interested in using behavioral types may still work on the same code base.

Overview Related work is discussed in Section 2. Our behavioral types are introduced in Section 3 together with behavioral runtime monitors. Section 4 describes the monitor integration with Java and AspectJ. Section 5 presents an example and a conclusion is given in Section 6.

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 that we also address in this paper. While the used formalism for expressing behavior in interface automata is more powerful (timed automata vs. automata vs. timing annotation per method), interface automata do not target the main focus of this paper: checking the behavior at runtime of a component by using some form of monitoring. They are especially aimed at compatibility checks of different components interacting at compile time of a system. The term behavioral types is used in the Ptolemy framework [12]. Here, the focus is on real-time systems.

The runtime verification community has developed frameworks which can be used for similar purpose as our behavioral type based monitors. The MOP framework [15] allows the integration of specifications into Java source code files and generates AspectJ aspects which encapsulate monitors. Compared to this work, the intended goals are different. While we keep the specification and implementation part separate, in order to be able to use the specification for different purposes at development, compile and runtime, a close integration of specification and code is often desired and realized in the runtime verification frameworks. A framework taking advantage of the trade-off between checking specifications at runtime and at development time has been studied in [9]. A framework that generates independent Java monitors leaving the instrumentation aspect to the implementation is described in [3]. Other topics explored in this context comprise, e.g., the efficiency and expressiveness of monitoring [2, 4] but are less focused on software engineering aspects compared to this paper.

Monitoring of performance and availability attributes of OSGi systems has been studied in [17]. Here, a focus is on the dynamic reconfiguration ability of OSGi. Another work using the .Net framework for runtime monitor integration is described in [11].

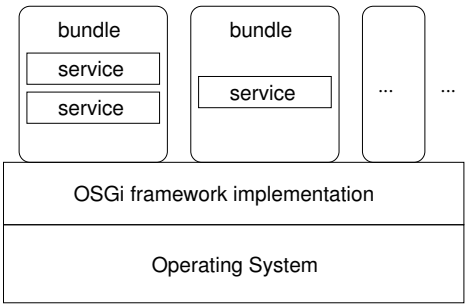

Fig. 1: OSGi framework

Runtime enforcement of safety properties was initiated with security automata [16] that are able to halt the underlying program upon a deviation from the expected behaviors. In our behavioral types framework, the enforcement of specifications is in parts left to the system developer, who may or may not take potential Java exceptions resulting from behavioral type violations into account.

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 [5]. Some investigations on the relation between OSGi and some more formal component models have been done in [13]. Aspects on formal security models for OSGi have been studied in [10].

3 Behavioral Types for OSGi

We present an overview on OSGi and describe our behavioral types. We present our vision for integrating behavioral types in the development and life-cycle of OSGi systems. Furthermore, we present the implemented generation of runtime monitors from behavioral types.

3.1 OSGi Overview

The OSGi framework is a component and service platform for Java. It allows the aggregation of services into bundles (cf. Figure 1) and provides means for dynamically configuring services, their dependencies and usages. It is 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 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. 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 and typically realized

by registering a service object implementing a special interface. 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[3] for Eclipse, Apache Felix[4] or Knopflerfish[5]. If bundles do not depend on implementation specific features, OSGi bundles can run on different implementations of the OSGi framework.

Services can run in parallel and are – if not explicitly synchronized – asynchronous. Method calls, even between objects in different bundles – are non-blocking. In the context of behavioral runtime monitoring using behavioral types, we are interested on how to monitor relevant semantics features of the runtime behavior rather than reasoning about the semantics features themself. For this paper, the monitoring of the order of method calls within and between components and their timing behavior and the dynamic creation and handling of monitors in accordance with the dynamic handling of bundles and objects are relevant.

3.2 Behavioral Types

Our behavioral types provide an abstract description of a components behavior and thus provide a way of formalizing specifications associated with the component. They can be used as a basis for checking the compatibility of components – for composing components into new ones, and interaction of different components – and for providing ways to make components compatible using coercion. Type conformance can be enforced at compile time (e.g., like primitive datatypes int and float in a traditional typing system) – if decidable and feasible – or at runtime of a system – e.g., like whether a pointer is assigned to an object of a desired type at runtime in a traditional typing system.

In our work behavioral types are realized as files that contain a description of (parts of the) behavior of an OSGi component. Typically, there should be one file per bundle, or class definition. But different aspects of behavior may also be realized using different files. In Eclipse the files are associated with an OSGi bundle by putting them in the same project folder in the Eclipse workspace. Here, behavioral types are formally defined using the following ingredients.

Behavioral Type Automaton A behavioral type automaton is a finite automaton represented as a tuple (Σ, L, l_0, E) comprising 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 $\sigma \in \Sigma$. A consistency condition on our types is that all $\sigma \in \Sigma$ appear in some transition in E.

[3] http://www.eclipse.org/equinox/
[4] http://felix.apache.org/site/index.html
[5] http://www.knopflerfish.org/

In this paper, since we are interested in method calls, Σ is the set of method names of components. The definition presented here can be used for specifying the behavior of single objects, all objects from a class, bundles and their interactions. It can be used for monitoring incoming method calls, outgoing method calls, or both.

Maximal Execution Time Table In addition to the protocol defined by the behavioral type automaton, we define the maximal execution time of methods as a mapping $\Sigma \rightarrow long \cup \perp$ from the set of method names Σ to their maximal execution time in milliseconds. The specification of a maximal execution time is optional, thus, the $\perp$ entry indicates that no maximal execution time is set.

The behavioral type automaton together with the corresponding maximal execution time table form a behavioral type. Additional descriptive information is optionally available, but not used for the behavioral runtime monitoring aspects that are described here. Other representations such as Σ comprising method signatures and timing information are possible future extensions.

3.3 Behavioral Types at Development and Runtime of a System

A potential major advantage of using behavioral types is the support of a seamless integration of behavioral specification throughout the development phase and the life cycle of a system. Our behavioral types can be used for different purposes (we proposed them in [8]) at development and runtime. A main idea of using behavioral types at development time is to derive them from requirements and use them for refinement checking of different forms of specification for the same entity that are supposed to have some semantical meaning in common. For example, the abstract specification, source code and compiled code of the same component represent different abstraction levels and should fulfill the same behavioral type. Checking this could be done by using static analysis at development time. At the end of a development process, a developed OSGi bundle is deployed including the behavioral type files. These can now be used for additional (dynamic) operations in the running system. Figure 2 shows two operations which can be carried out at runtime of a system: the registration and discovery of components using the OSGi framework, the compatibility, e.g., deadlock checking of bundle interaction protocols. Behavioral runtime monitors and their derivation from the development process are shown in Figure 3. The Figure shows the generation of the behavioral runtime monitor and its connection using aspects at development time and the actual monitoring at runtime. Up till now, we have implemented editors, registration of OSGi components, compatibility (deadlock freedom) of specifications, some form of dynamic adaptation as proposed in [6], and the behavioral runtime monitors which are new to this paper.

3.4 Monitor Generation

Regardless of what we intend to monitor, the monitor generation from a specification is the same. It is done automatically from a behavioral type file and generates a single Java file that defines a single monitor class.

Figure 4 shows a generated monitor. Monitors are generated as classes bearing a name derived from the original behavioral type. They comprise a map `maxtimes` that

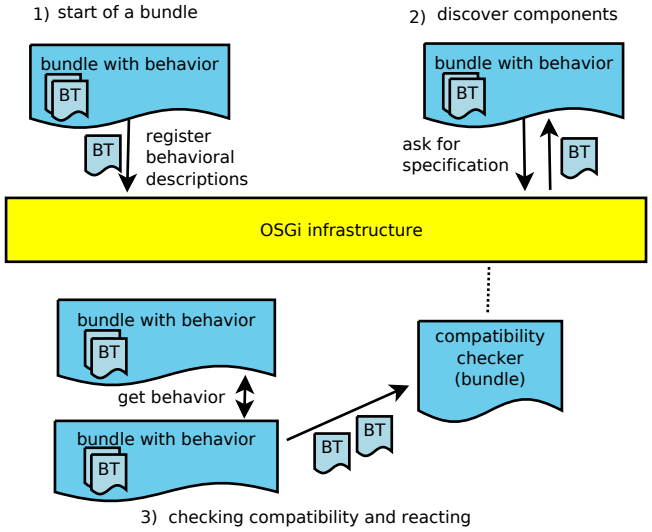

Fig. 2: Behavioral types at runtime

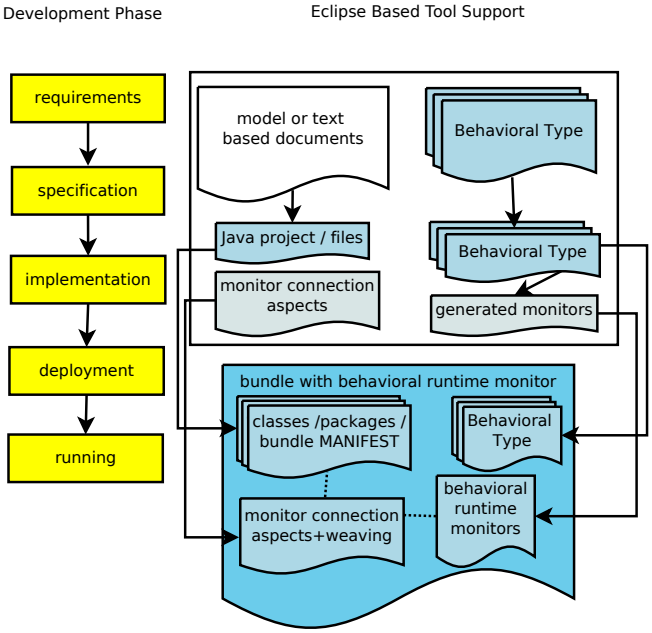

Fig. 3: Derivation of behavioral runtime monitors

maps method names to their maximal execution time in milliseconds. This entry is optional. If present, this map is initialized by the constructor –

`public clientinstance_out_realistic_simple_mon()`

in the example – of the monitor with the values specified for methods in the behavioral type file. Generated from an automaton from the behavioral type our behavioral runtime monitors comprise a static enumeration type with the location names of the automaton. In the automaton, the locations `LOCs0, LOCs1` are present. Using this type a state transition function generated from the transition relation is generated. The state transition function takes a string encoding a method name – event name – and updates a state field `protected LOCATION state` of the method. This field is initialized on object creation with the name of the initial state: `LOCs0` in the example.

```
package monitors;
import ....
public class clientinstance_out_realistic_simple_mon {
   public Map<String,Long> maxtimes = new HashMap<String,Long>();
   public clientinstance_out_realistic_simple_mon() {
        maxtimes.put("listFlights",new Long(1000)); }
   public static enum LOCATION {  LOCs0 , LOCs1 }
   protected LOCATION state = LOCATION.LOCs0;
   public boolean nextState(String event) {
      boolean rval = false;
      switch (state) {
         case LOCs0:
            ...
            break;
         case LOCs1:
            if (event.equals("listFlights")) {
               state = LOCATION.LOCs1;
               rval = true;
            }
            ....
            if (event.equals("listFlight")) {
               state = LOCATION.LOCs1;
               rval = true;
            }
            break;
      }
      return rval;
   }
}
```

Fig. 4: Generated example monitor

4 Behavioral Runtime Monitor Integration using AspectJ

The generated monitors are connected to the component that shall be observed using AspectJ aspects. AspectJ is an extension of Java that features aspect oriented programming. Aspects are specified in separate files and feature pointcuts that allow the specification of locations where Java code specified in the aspect shall be added to existing Java code. This weaving of aspect code into existing Java code is done on bytecode level.

Monitors are created and called from aspects. All extra code needed to integrate the monitors is defined in the AspectJ files or in libraries accessed through the AspectJ files. There is no need to touch the source code of a component. This independence of source code and specification is a design goal of our framework. We distinguish different kinds of monitor deployment. Each kind requires its own aspect template and its instantiation.

Singleton monitors In some cases it is sufficient to use a singleton instance of a monitor. This is the case when monitoring all the method calls that occur in a bundle, within all objects of a class, or within a singleton object. For monitoring method call orders, we use a `before` pointcut in AspectJ. Figure 5 shows an example aspect: Here, before the calls to methods – specified in the execution pattern after the ":" in the pointcut – of all objects of class `MiddlewareProc` an update on the state transition function – the `com.nextState` – is inserted. We extract the name of the called method using reflection and a helper method `AJMonHelpers.getMethodName` and pass it to the state transition function. In addition to updating the state field in the monitor we get a boolean value indicating whether the monitored property is still fulfilled. In case of a deviation the `BehavioralTypeViolationException` – a runtime exception is thrown. The implementation of the `MiddlewareProc` class may or may not catch this exception and react to it.

```
package bookingsystem.middleware;
import java.util.HashMap;
import java.util.Map;
....
import monitors.*;

public aspect CallincprotocolMiddlewareProc {
    ...
    pointcut myMethod(MiddlewareProc p): this(p) &&
       within(MiddlewareProc) && execution(* *(..));
    before (MiddlewareProc p): myMethod(p) {
        ...
     boolean verdict = com.nextState(
           AJMonHelpers.getMethodName(
              thisJoinPointStaticPart.getSignature()));
     if (!verdict) throw new BehavioralTypeViolationException();
    }
}
```

Fig. 5: Example aspect

Multiple monitor instances In same cases we want to monitor each object of a class with an independent monitor. Here, we create on call of the object's constructor an individual monitor for the object. It is added to a (hash)map (Object $\rightarrow$ Monitor). Since the AspectJ pointcuts are defined with respect to the static control flow information specified in the source code of a class, on each call of a method belonging to the class to be monitored, we use the same code in each object and chose the monitor for the particular object by looking it up from the map and advance the respective monitor state.

Monitoring of time Monitoring time is done using Java timers within the Java code associated with the pointcuts. On call of a method we create a timer that is scheduled to throw an exception after the specified maximal execution time. Using the `after` pointcut, the timer is canceled if the method's execution finishes on time and thus, no exception is thrown in this case.

The adaptation of an aspect for monitoring a particular component is simple. One has to take the appropriate AspectJ .aj file and adapt it, by inserting the names of the classes and packages that shall be monitored and the correct monitor names. Weaving of the aspects is done automatically on Java bytecode level and no additional configuration needs to be done.

5 Example and Evaluation

One example scenario regarded by us is the flight booking system (our set-up comprises only the functionality necessary for our monitoring experiments) shown in Figure 6. OSGi components and their interactions are shown. Clients are represented as proxy components in the system and served by middleware processes which are created and managed by a coordination process. Middleware processes use concurrently a flight database and a payment system which are represented by proxy OSGi components. We have investigated the communication structure between the components and investigated deployment of monitors. This comprises the following cases: 1) The use of multiple monitors running in parallel and being created at runtime for different objects which are created dynamically. In the example system this is the case for the middleware processes, where processes are created as separate objects on demand and are monitored independently of each other. 2) The monitoring of all objects of a single class using a single monitor and the monitoring of singleton objects and the monitoring of bundle behavior. This is, e.g., the case in the payment subsystem. 3) Furthermore, we have investigated the monitoring of maximal execution time of methods. In the example system this is the case in the payment subsystem and access to the flight database. We did not find any major problems in our approach.

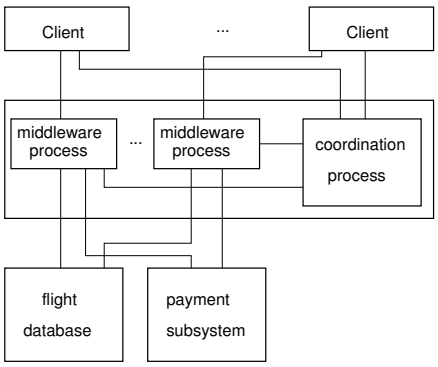

Fig. 6: Components of our flight booking system

6 Conclusion

We presented work on behavioral types for Java / OSGi components and the monitoring of behavioral type based specifications at runtime of a system. Our Eclipse based implementation allows the behavioral runtime monitoring of components without modifying their source code by using aspect oriented programming. In addition to the behavioral runtime monitoring work, the same behavioral types can be used for other operations at compile time, e.g., static analysis of component compatibility, and runtime, e.g., discovery of components in a SOA like scenario, dynamic adaptation of components [6].

References

1. L. de Alfaro, T.A. Henzinger. Interface automata. Symposium on Foundations of Software Engineering, ACM , 2001.
2. H. Barringer, Y. Falcone, K. Havelund, G. Reger, D. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. Formal Methods, vol. 7436 of LNCS, Springer-Verlag, 2012. (FM'12)
3. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Verification, Model Checking, and Abstract Interpretation, vol. 2937 of LNCS, Springer-Verlag, 2004. (VMCAI'04)
4. Bauer, A., Leucker, M.: The theory and practice of SALT. NASA Formal Methods, vol. 6617 of LNCS, Springer-Verlag, 2011.
5. J. Bauml and P. Brada. Automated Versioning in OSGi: A Mechanism for Component Software Consistency Guarantee. Euromicro Conference on Software Engineering and Advanced Applications, 2009.
6. J. O. Blech, Y. Falcone, H. Rueß, B. Schätz. Behavioral Specification based Runtime Monitors for OSGi Services. Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), vol. 7609 of LNCS, Springer-Verlag, 2012.
7. J. O. Blech, H. Rueß, B. Schätz. On Behavioral Types for OSGi: From Theory to Implementation. http://arxiv.org/abs/1306.6115. arXiv.org 2013.
8. J. O. Blech and B. Schätz. Towards a Formal Foundation of Behavioral Types for UML State-Machines. UML and Formal Methods. ACM SIGSOFT Soft. Eng. Notes, 2012.
9. E. Bodden, L. Hendren. The Clara framework for hybrid typestate analysis. Software Tools for Technology Transfer (STTT), vol. 14, 2012.
10. O. Gadyatskaya, F. Massacci, A. Philippov. Security-by-Contract for the OSGi Platform. Information Security and Privacy Conference, IFIP Advances in Information and Communication Technology, vol. 376, 2012.
11. K.W. Hamlen, G. Morrisett, F.B. Schneider. Certified in-lined reference monitoring on .NET. Programming languages and analysis for security, ACM 2006.
12. E.A. Lee, Y. Xiong. A behavioral type system and its application in ptolemy ii. Formal Aspects of Computing, 2004.
13. M. Mueller, M. Balz, M. Goedicke. Representing Formal Component Models in OSGi. Proc. of Software Engineering, Paderborn, Germany, 2010.
14. OSGi Alliance. OSGi service platform core specification (2011) Version 4.3.
15. P. O'Neil Meredith, D. Jin, D. Griffith, F. Chen, G. Roşu. An Overview of the MOP Runtime Verification Framework. Software Techniques for Technology Transfer, Springer, 2011.
16. F.B. Schneider. Enforceable security policies. ACM Transactions on Information and System Security, vol. **3**, ACM, 2000.
17. F. Souza, D. Lopes, K. Gama, N. Rosa, R. Lima. Dynamic Event-Based Monitoring in a SOA Environment. On the Move to Meaningful Internet Systems, vol. 7045 of LNCS, Springer-Verlag, 2011.