Coordination of Software Components with BIP: Application to OSGi

Simon Bliudze
Rigorous System Design Laboratory EPFL
1015 Lausanne, Switzerland
simon.bliudze@epfl.ch

Anastasia Mavridou
Rigorous System Design Laboratory EPFL
1015 Lausanne, Switzerland
anastasia.mavridou@epfl.ch

Radoslaw Szymanek
Crossing-Tech S.A.
1015 Lausanne, Switzerland
radoslaw.szymanek@crossing-tech.com

Alina Zolotukhina
Rigorous System Design Laboratory EPFL
1015 Lausanne, Switzerland
alina.zolotukhina@epfl.ch

ABSTRACT

Coordinating component behaviour and access to resources is among the key difficulties of building large concurrent systems. To address this, developers must be able to manipulate high-level concepts, such as Finite State Machines and separate functional and coordination aspects of the system behaviour. OSGi associates to each bundle a state machine representing the bundle's lifecycle. However, once the bundle has been started, it remains in the state `Active`— the functional states are not represented. Therefore, this mechanism is not sufficient for coordination of active components.

In this paper, we present a methodology for functional component coordination in OSGi by using BIP coordination mechanisms. BIP allows us to clearly separate the system-wide coordination policies from the component behaviour and the interface that components expose for interaction. By using BIP, we show how the allowed global states and state transitions of the modular system can be taken into account in a non-invasive manner and without any impact on the technology stack within an OSGi container.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*Methodologies*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and interfaces, State diagrams*

General Terms

Design, Reliability

Keywords

BIP, OSGi, Component coordination, Concurrency

MiSE '14, June 2 - June 3, 2014, Hyderabad, India

1. INTRODUCTION

When building large concurrent systems, one of the key difficulties lies in coordinating component behaviour and, in particular, concurrent access to resources. Native mechanisms such as, for instance, locks, semaphores and monitors allow developers to address these issues. However, such solutions are complex to design, debug and maintain. Indeed, coordination primitives are mixed up with the functional code, forcing developers to keep in mind both aspects simultaneously. Finally, in concurrent environments, it is difficult to envision all possible execution scenarios, making it hard to avoid common problems such as race conditions.

The coordination problem above calls for a solution that would allow developers to think on a higher abstraction level, separating functional and coordination aspects of the system behaviour. For instance, one such solution is the AKKA library [15] implementing the Actor model. An actor is a component that communicates with other components by sending and receiving messages. Actors process messages atomically. The state of an actor cannot be directly accessed by other actors, avoiding such common problems as data races. However, component coordination and resource management are still difficult. Fairly complex message exchange protocols have to be designed, which are spread out across multiple actors. Any modification of the coordination policy calls for corresponding modifications in the behaviour of several actors, potentially leading to cascading effects and rendering the entire process highly error-prone.

Our approach relies on the observation that the behaviour of a component can be represented as a Finite State Machine (FSM). An FSM has a finite set of states and a finite set of transitions between these states. Transitions are associated to functions, which can be called to *force* a component to take an action or to *react* to external events coming from the environment. Such states and transitions usually have intuitive meaning for the developer. Hence, representing components as FSMs is a good level of abstraction for reasoning about their behaviour. In our approach, the primitive coordination mechanism is the synchronisation of transitions of several components. This mechanism gives the developers a powerful and flexible tool to manage component coordination. This allows a clear separation between the component

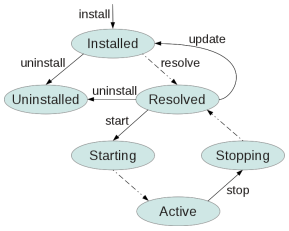


Figure 1: Bundle lifecycle in OSGi.

behaviour and system-wide coordination policies.

For coordination scenarios that require a global state information, dedicated *monitor* components can be added in straightforward manner. This allows to centralise all the information related to the coordination in one single location, instead of distributing it across the components. Furthermore, it considerably simplifies the system maintenance and improves reusability of components. Indeed, components do not carry coordination logic based on the characteristics of any specific execution environment.

An observable trend in software engineering is that design becomes more and more declarative. Developers provide specifications of *what* must be achieved, rather than *how* this must be achieved. These specifications are then interpreted by the corresponding engines, which generate — often on the fly — the corresponding software entities. Thus, it is not always possible to instrument or even access the actual source code. Furthermore, it is usually not desirable to modify such code, since this can lead to a considerable increase of the maintenance costs. Our approach is based on a non-invasive mechanism relying, for the interaction with the controlled components, on existing API.

We present a methodology for functional component coordination by using BIP coordination mechanism and its implementation based on OSGi that:

- allows clear separation between component behaviour and system-wide coordination policies;
- improves component usability and simplifies system maintenance;
- does not require access to the source code.

The paper is structured as follows. Section 2 provides background information about OSGi and BIP. Section 3 presents the models of the Camel routes use case. Section 4 shows how the proposed methodology is applied in practice. Section 5 describes the implemented software architecture. Section 6 discusses related work and Section 7 concludes the paper.

2. BACKGROUND

2.1 The OSGi standard

OSGi defines an architecture for developing and deploying modular applications [18], which are assembled from multiple components, called *bundles*. Bundles are comprised of Java classes and resources. OSGi controls the lifecycle of a bundle, its installation, starting, stopping, updating and deinstallation, on the fly without the need to restart the system. Furthermore, OSGi manages the resolution of bundle dependencies, versioning and classpath control. Bundles provide classes and *services* that can be used by other bundles. A service is defined by the Java interface it implements.

A state machine representing the bundle's lifecycle (Figure 1) is associated to each bundle. A bundle can be in one of the states `Installed`, `Resolved`, `Active`, etc. However, once the bundle is started, it remains in the state `Active`— the functional states are not represented. Hence, this mechanism is not applicable for coordination of active components.

2.2 The BIP framework

BIP [6] is a component framework encompassing rigorous system design. In BIP, systems are constructed by superposing three layers of modelling: *Behaviour*, *Interaction*, and *Priority*. Behaviour consists of a set of components modelled by FSMs that have transitions labelled with *ports*. Ports form the interface of a component and are used to define interactions with other components.

The second layer models interaction between components. Interaction models define allowed synchronization between components and can be represented in many equivalent ways. Among these are the Algebra of Connectors [10] and boolean formulæ on variables representing port participation in interactions [11]. Connectors are most appropriate for graphical design and interaction representation, whereas boolean formulæ are most appropriate for manipulation and efficient encoding. When several interactions are possible, priorities can be used as a filter. Interaction and Priority layers are collectively called *glue*. In this paper, we only consider interaction models and leave priorities for future work.

The BIP framework comprises a language and an associated tool-set supporting the Rigorous Design Flow [6]. The BIP language allows building complex systems by specifying the coordination between a set of atomic components. The execution of a BIP system is driven by the BIP Engine applying the following protocol in a cyclic manner:

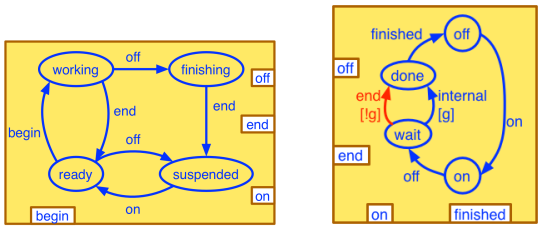
1. Upon reaching a state, each component notifies the BIP Engine about the possible outgoing transitions;
2. The BIP Engine picks one interaction satisfying the glue specification and notifies the involved components;
3. The notified components execute the functions associated to the corresponding transitions.

Our work is based on implementing functional component coordination by using the BIP coordination mechanisms. However, the BIP language and tools mentioned above are not used in this approach.

3. CAMEL ROUTES USE CASE

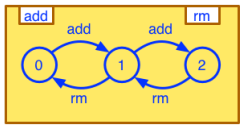
Using BIP allows taking into consideration the structure of the controlled software and the coordination constraints imposed by the safety properties. BIP coordination extension for OSGi has been implemented and tested in Connectivity Factory^{TM}[1], the flagship product of Crossing-Tech S.A. The main use-case consists in managing memory usage by a set of Camel routes[2]. A Camel route connects a number of data sources to transfer data among them. Data can be fairly large, may require additional processing and thus, routes share and compete for memory. Without additional coordination, simultaneous execution of several Camel

[1] `http://www.crossing-tech.com/`
[2] `http://camel.apache.org/routes.html`



(a) Camel Route

(b) Switchable Route



(c) Monitor

Figure 2: Examples of component models.

routes can lead to `OutOfMemory` exceptions, even when each route has been tested and sized appropriately on its own. For the sake of simplicity, we assume that all active routes consume the same amount of memory. Thus, it is sufficient to ensure that the number of active routes does not exceed a given threshold.

We have designed BIP models for Camel routes, using the `suspend` and `resume` functions provided by the Camel Engine API. By introducing an additional Monitor component, we limit the number of routes running simultaneously to ensure that the available memory is sufficient for the safe functioning of the system.

The model of a Camel route is shown in Figure 2(a). It has four states: `suspended`, `ready`, `working` and `finishing`. The route has transitions `begin` and `end` between the states `ready` and `working`, corresponding to processing a file. The route can be turned off through the `off` transition. Figure 2(b) presents a different model of the route. It is obtained by merging the `ready` and `working` states together (the `on` state), and by splitting the `finishing` state into two: `wait` and `done`. We use notification policies provided by Camel to observe spontaneous modifications of the route states. This differentiates the types of transitions used in the model, which is further explained in the next section.

In Figure 2(c) the model of the Monitor is presented. It has three states that correspond to the number of simultaneously active Switchable routes, which is limited to two. This is achieved by enforcing, for each route, the synchronisation between its port `on` (respectively `finished`) and the port `add` (respectively `rm`) of the Monitor through the use of BIP connectors. It is worth noticing that Monitor components carry only coordination logic but do not contain any functional code. To reuse the Monitor component for a larger number of routes, the model has to be extended with more states. Next section presents how the component models and their coordination constraints can be specified.

4. DESIGN METHODOLOGY

4.1 Component model

A component is represented by an FSM extended with ports. The FSM is specified by its states and guarded transitions. Each transition has a function and an associated port. One port can be associated to several transitions. The firing of a single transition happens as follows:

1. The transition is checked for enabledness: a transition is enabled when it has no guard or when its guard evaluates to true. Only enabled transitions can be fired;
2. The function associated with the transition is called;
3. The current state of the FSM is updated.

We define three types of transitions: *internal*, *spontaneous* and *enforceable*. Internal transitions represent computations independent of the environment and are used to make the component models more concise; when enabled, they are executed immediately. Spontaneous transitions represent changes in the environment that affect the component behaviour but are not controlled. Enforceable transitions represent the controllable behaviour of the component. To ensure execution determinism, at most one internal transition can be enabled at any execution step. The type of a transition is inferred from the type of the port associated to it.

Transitions of different types are illustrated in Figure 2(b). Firing the enforceable transition `off` takes the route into state `wait`, from which two transitions are possible, both leading to the state `done`. The internal transition can be taken if the route has finished processing the files (the associated guard `g` is satisfied). Otherwise, the component waits for the notification of the spontaneous event `end`.

In our implementation, the execution of transitions is controlled and managed by a *BIP Executor* object, while the synchronisation between components is orchestrated by a dedicated *BIP Engine*. BIP Executor maintains a queue of notifications corresponding to spontaneous transitions and cyclically executes the following two steps: first, all transitions from the current state are checked for enabledness; and second, one transition is picked for execution. Transition choice depends on its type (in order of decreasing priority):

1. If an internal transition is enabled, it is fired.
2. If a spontaneous transition is enabled and a corresponding notification is available in the queue, this spontaneous transition is fired. If there are no notifications in the queue corresponding to enabled spontaneous transitions and no enforceable transitions are enabled, the component waits for the first notification of one of the enabled spontaneous transitions.
3. If enforceable transitions are enabled, the Executor informs the Engine about the current state and the enabled ports (i.e. which transitions can be performed). The Executor then waits for a response from the Engine indicating the port to execute. Upon receiving this response, it performs the corresponding transition.

When a spontaneous and an enforceable transition are enabled simultaneously, but a notification corresponding to the former has not been received yet, the BIP Executor announces the latter to the BIP Engine. Even if a notification arrives, it will not be processed until the next cycle.

4.2 Design steps

In our approach the developer does not need to access or modify existing source code. Designers provide specifications as separate files and are responsible for enforcing

```
@bipComponentType(initial = "off",
                  name = "SwitchableRoute")
@bipPorts({ @bipPort(name = "end", type = "spontaneous"),
            @bipPort(name = "on", type = "enforceable"),
            @bipPort(name = "off", type = "enforceable"),
            @bipPort(name = "finished",
                     type = "enforceable")})

public class SwitchableRoute {
  ...
  @bipTransition(name = "end", source = "wait",
              target = "done", guard = "!g")
  public void spontaneousEnd() {/* method body */}

  @bipTransition(name = "finished", source = "done",
                target = "off", guard = "")
  public void finishedTransition() {/* method body */}

  @bipGuard(name = "g")
  public boolean isFinished() {/* method body */}
  ...}
```

Figure 3: Annotations for the Switchable Route.

their validity. However, our tools log an exception when a non-valid specification is detected at runtime.

The design process involves two steps: first, defining the behaviour for each component; and second, specifying their interaction constraints.

4.2.1 *Specification of component behaviour*

An FSM extended with ports, as presented in Section 4.1, is provided as an instance of a Java class implementing the dedicated `Behaviour` interface, which is used by the Executor and the Engine and provides access to the information about states, ports, transitions and guards of the FSM.

Developers need to provide the following: the name of the component; the list of all states and the initial state; the list of ports; the list of transitions with corresponding guards; and a reference to the object implementing the methods associated to guards and transitions of the component.

There are two different ways to provide this information. One way is to build such objects from specification *annotations*. Annotations are syntactic metadata associated to parameters, fields, methods or class declarations. Annotations in a BIP Specification are processed by the BIP Executor, which we have developed as part of our library, to construct a corresponding Behaviour object representing the FSM. For specifying larger components, where annotations become impractical, we have defined a Behaviour API, which allows developers to construct a Behaviour object in a programmatical manner.

Figures 3 and 4 partly present the annotated Java classes for the components of Figures 2(b) and 2(c) respectively. We have defined the following annotations:

`@bipComponentType` — associated to a BIP component specification class. It has two fields: `name` of the component type, and `initial` state.

`@bipPort` — associated to a BIP component specification class. It has two fields: port `name`, and port `type` which can be "spontaneous" or "enforceable", specifying the type of the transitions. The internal transitions are defined by omitting their name. We use an additional `@bipPorts` annotation to specify several instances of the `@bipPort` annotation.

`@bipGuard` — associated to a method that computes a

```
...
@bipTransitions({
@bipTransition(name = "add", source = "0",
 target = "1", guard = ""),
@bipTransition(name = "add", source = "1",
 target = "2", guard = "")})
public void addRoute() {routeCounter++;}
...
```

Figure 4: Annotations for the Monitor.

```
<glue>
    <accepts>
        <accept>
            <effect id="on" specType="SwitchableRoute"/>
            <causes>
               <port id="add" specType="Monitor"/>
            </causes>
        </accept>
    </accepts>
    <requires>
          <require>
            <effect id="on" specType="SwitchableRoute"/>
            <causes>
                <port id="add" specType="Monitor"/>
            </causes>
        </require>
    </requires>
</glue>
```

Figure 5: Interaction constraints.

transition guard. The method must return a boolean value. This annotation has one field: guard `name`.

`@bipTransition` — associated to a transition handler method with four fields: port `name` labelling the transition, `source` state, `target` state and `guard`, a boolean expression on the names of the guards (true if omitted). The guard expressions can be specified using parenthesis and basic logical operators. Only ports defined by the `@bipPort` annotation can be used as transition names.

4.2.2 *Specification of interaction constraints*

To define the interaction model, the developer specifies the interaction constraints of each component. An interaction constraint can be provided for each port of a system. Two types of constraints are used to define allowed interactions:

Causal constraints (`Require`): used to specify ports of other components that are necessary for any interaction involving the port to which the constraint is associated.

Acceptance constraints (`Accept`): used to define optional ports of other components that are accepted in the interactions involving the port to which the constraint is associated.

For example, the constraint `SwitchableRoute.on Require Monitor.add` forces the port `on` of any component of type `SwitchableRoute` to synchronise with a port `add` of some component of type `Monitor`. Furthermore, the constraint `SwitchableRoute.on Accept Monitor.add` specifies that no other ports are allowed to participate in the same interaction. In the current implementation interaction constraints are given in an XML file (cf. Figure 5).

5. IMPLEMENTATION

The software architecture of the proposed framework is shown in Figure 6. The architecture consists of two major parts: the part that involves the components to control (on

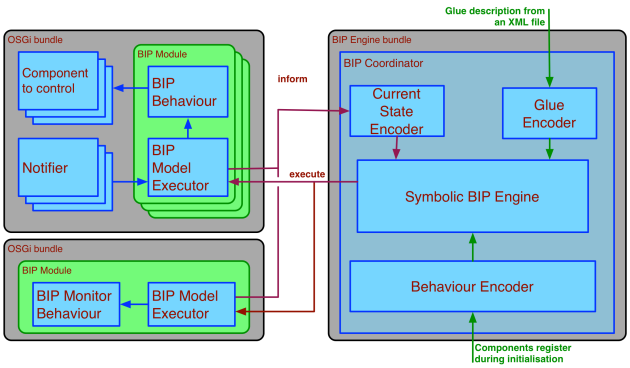


Figure 6: Software architecture.

```
******************** Inform ********************
Component: switchableRoute395 is at state: done
Component: switchableRoute393 is at state: off
Component: switchableRoute394 is at state: done
Component: monitor396 is at state: 2
******************** Engine ********************
ChosenInteraction:
Component: switchableRoute395 with port: finished
Component: monitor396 with port: rm
******************** Inform ********************
Component: switchableRoute395 is at state: off
Component: switchableRoute393 is at state: off
Component: switchableRoute394 is at state: done
Component: monitor396 is at state: 1
```

Figure 7: BIP Engine: one execution cycle printout.

the left) and the BIP Engine part (on the right). The grey outer boxes in the diagram represent OSGi bundles.

For each component a BIP Behaviour is generated at runtime, which contains information about the FSM and ports of the controlled entities. As shown in Figure 6, each instance of BIP Behaviour is coupled with a dedicated instance of BIP Executor to form a *BIP Module*. The *Notifier* is an additional component that informs the BIP Executor of spontaneous events relevant to the controlled entities.

The BIP Executor builds the Behaviour object from the component specification or receives it as an argument at initialisation (as specified in Section 4.2.1). BIP Executor implements the protocol presented in Section 4.1. In order to ensure consistency of guard valuations, guard functions are computed only once at each execution cycle, regardless of how many times they are used. At each execution cycle, the Executor interprets the Behaviour and fires the transitions, invoking the methods using Java Reflection mechanism [14].

The operational semantics of BIP is implemented by a dedicated Engine (presented in the right part of Figure 6), which is used for the coordination of software modules according to the three-step protocol presented in Section 2.2. The implementation of the Engine is modular and consists of five main parts: three Encoders, the BIP Coordinator and the Symbolic BIP Engine. It uses Binary Decision Diagrams (BDDs)[3] [3], that are efficient data structures to store and manipulate boolean formulæ.

The ports and states of the system components are associated to boolean variables. These boolean variables are used at initialization by the *Behaviour* and *Glue Encoders* to translate the behaviour and glue constraints (`glue description` and `register` arrows in Figure 6) into boolean formulæ. During runtime, each component provides information about its current state and enabled ports (`inform` arrow). This information is also translated into boolean formulæ by the *Current State Encoder*.

At each execution cycle, the *Symbolic BIP Engine* computes the conjunction of these constraints to obtain the global boolean formula that represents the possible interactions of the system. It then chooses one interaction, notifies the *BIP Coordinator* that orders the components to make the necessary transitions (`execute` arrow). The BIP Coordinator manages the flow of information between the components and the Symbolic Engine through a dedicated `BIP Engine` interface. The BIP Engine is packaged as an OSGi bundle, using the mechanisms provided by OSGi to publish the service that can be used by other software modules.

An example of the Engine execution cycle is shown in Figure 7 for the Camel route use case (cf. Section 3) with three Switchable routes and one Monitor with component IDs 393–396 respectively. Two out of three Switchable routes inform the BIP Engine that they are at state `done` and therefore the Monitor correctly informs the BIP Engine that it is at state two, which corresponds to the number of active routes. In the next step, the BIP Engine selects the interaction `finished · rm`, which forces the Monitor to decrement the counter due to the completion of the ID 395 route.

6. RELATED WORK

Different approaches have been proposed to deal with the coordination of concurrent systems. First of all, locks and semaphores [16] have been extensively used in software engineering approaches to address concurrency problems. However, these solutions do not allow a clear separation between the functional code and the coordination mechanisms, making it hard to design and maintain correct programs, especially when they are used in large concurrent systems.

A Coordinator service developed by the OSGi community [19] allows simple coordination between multiple software components. A Coordinator object has only two states: Active and Terminated. This approach provides a higher abstraction level primitive for multi-party synchronisation barriers. Thus, some simple coordination can be ensured for several entities having no information about each other.

A different approach is taken by the AKKA library [15], that implements the Actor model [2]. Actors are concurrent components that communicate asynchronously through ports and avoid the use of low-level primitives, such as locks and semaphores. However, component coordination through complex message exchange protocols among multiple actors can be challenging and error prone.

Apart from BIP, the most prominent component-based frameworks found in the literature are Ptolemy [13] and Reo [4]. In particular, Reo is a channel-based exogenous coordination model wherein complex coordinators, called connectors, are compositionally built out of simpler ones to orchestrate component instances in a component-based system. The Ptolemy framework [13] adopts an actor-oriented view of a system. Ptolemy actors can be hierarchically in-

[3]We have used the JavaBDD decision diagram package available at `http://javabdd.sourceforge.net/`.

terconnected and support heterogeneous, concurrent modeling and design. For both of these frameworks, we are not aware of work on using coordination models to control the behaviour of pre-existing independently developed software components and, in particular, OSGi bundles.

A number of approaches have been proposed to the specification of OSGi component behaviour. In particular, Blech et al. [7] propose a framework to describe behavioural specification of OSGi bundles that can be used for runtime verification. The proposed semantics bears similarity to the semantics of the BIP framework [6]. Runtime checks are performed using constraint specifications to ensure safety in case of deviation from the original specification. The behavioral models of the components are loaded from XML files and integrated into a bundle [8]. The runtime monitors used are connected to the components by using AspectJ [8]. The aspects are specified in separate files and have pointcuts that define the locations where additional code must be added to the existing one. This approach requires detailed knowledge of the source code, whereas our approach relies only on the knowledge of the APIs provided by the components.

Another approach for OSGi-based behaviour specification has been studied by Mekontso Tchinda et al. [17]. The authors propose specifying OSGi services based on a combined use of interface automata [12] and process algebra [5]. Their specification of services is centered on finding the best candidates for service substitution. Qin et al. [20] propose a framework that specifies the behaviour of OSGi components through the use of WF-nets [1]. In their approach, behaviour description files are used to specify not only the service behaviour but also coordination protocols.

7. CONCLUSIONS

In this paper, we presented our approach to adding BIP coordination to OSGi. We described the architecture of the implemented framework. This architecture relies on several architectural elements, in particular a dedicated BIP Engine and a BIP Module. The latter comprises an annotated Java class, called BIP Specification, interpreted by an associated BIP Executor object. Our implementation of the BIP Engine is itself modular. It relies on a symbolic kernel manipulating boolean formulæ and three encoders that translate component and glue specifications into such formulæ. We also presented a use case illustrating our approach.

The full version of the paper is available as a technical report [9]. We consider that our work, recognizing the fact that bundles may have multiple components with multiple functional states, will help to improve the OSGi standard. However, our work can be used in coordinating any software components and need not be restricted to OSGi context.

Ongoing and future work consists in implementing data transfer mechanism between components, priority models and taking into account dynamically evolving system architectures where components can arrive and disappear.

8. ACKNOWLEDGMENTS

This work was partially supported by the Swiss Commission for Technology and Innovation (CTI 14432.1 PFES-ES).

9. REFERENCES

[1] W. Aalst. Verification of workflow nets. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 407–426. Springer, 1997.

[2] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, MA, USA, 1986.

[3] S. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, 1978.

[4] F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[5] J. C. M. Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, 335(2-3):131–146, May 2005.

[6] A. Basu, S. Bensalem, M. Bozga, et al. Rigorous Component-Based System Design Using the BIP Framework. *IEEE Software*, 28(3):41–48, 2011.

[7] J. O. Blech, Y. Falcone, H. Rueß, and B. Schätz. Behavioral specification based runtime monitors for OSGi services. In *ISoLA*, pages 405–419, Berlin, Heidelberg, 2012. Springer-Verlag.

[8] J. O. Blech, H. Rueß, and B. Schätz. On behavioral types for OSGi: From theory to implementation. *CoRR*, abs/1306.6115, 2013.

[9] S. Bliudze, A. Mavridou, R. Szymanek, and A. Zolotukhina. Integration of BIP into Connectivity Factory: Implementation. Technical report, 2013. https://infoscience.epfl.ch/record/196996.

[10] S. Bliudze and J. Sifakis. The algebra of connectors—structuring interaction in BIP. *IEEE Transactions on Computers*, 57(10):1315–1330, 2008.

[11] S. Bliudze and J. Sifakis. Synthesizing Glue Operators from Glue Constraints for the Construction of Component-Based Systems. In S. Apel and E. Jackson, editors, *Software Composition*, LNCS, pages 51–67, Berlin / Heidelberg, 2011. Springer.

[12] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-9, pages 109–120, New York, NY, USA, 2001. ACM.

[13] J. Eker, J. Janneck, E. Lee, et al. Taming heterogeneity: The Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[14] I. R. Forman, N. Forman, D. J. V. Ibm, I. R. Forman, and N. Forman. Java reflection in action, 2004.

[15] M. Gupta. *Akka Essentials*. Community experience distilled. Packt Publishing, 2012.

[16] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 1999.

[17] H. A. Mekontso Tchinda, N. Stouls, and J. Ponge. Spécification et substitution de services OSGi. Rapport de recherche RR-7733, INRIA, Sept. 2011.

[18] OSGi Alliance. *OSGi service Platform Core Specification*, Apr. 2007. Release 4, Version 4.15.

[19] OSGi Alliance. *Coordinator service*, `http://www.osgi.org/javadoc/r5/enterprise/org/osgi/service/coordinator/Coordinator.html`. (Accessed on 18/02/2014.).

[20] Y. Qin, H. Hao, L. Jim, G. Jidong, and L. Jian. An approach to ensure service behavior consistency in OSGi. In *APSEC*, 2005.