PROJECT: CTI 14432.1 PFES-ES

Integration of BIP into Connectivity Factory: Implementation

Project report: 2nd phase

Simon Bliudze, Anastasia Mavridou, Radoslaw Szymanek and Alina Zolotukhina

08/11/2013

Abstract:
Coordinating component behaviour and, in particular, concurrent 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 simple 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.
This report presents the methodology, proposed in the project, for functional component coordination in OSGi by using BIP coordination mechanisms. In BIP, systems are constructed by superposing three layers of modelling: *Behaviour*, *Interaction*, and *Priority*. This approach 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 have shown 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. We illustrate our approach on two use-cases, whereof one is based on a real-life application.

Table of Contents

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 [14] implementing the Actor model. An actor is a component that communicates with other components by sending and receiving messages. The processing of a message by an actor is atomic. 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 still spread out across multiple actors. Any modification of the coordination policy calls for the 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 primitive mechanism gives the developers a powerful and flexible tool to manage component coordination. Furthermore, this approach allows us to clearly separate the system-wide coordination policies from the component behaviour and the interface that components expose for interaction.

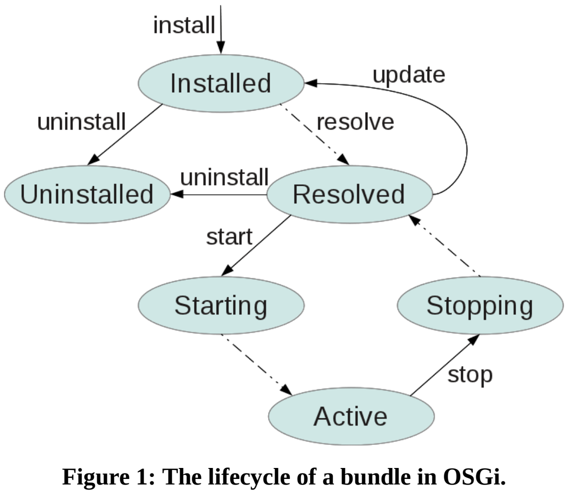


Figure 1: The lifecycle of a bundle in OSGi.

OSGi [17] associates to each bundle a simple state machine representing the bundle's lifecycle (Figure 1). A bundle can be in one of the states `Installed`, `Resolved`, `Active`, etc. However, once the bundle has been started, it remains in the state `Active`—the functional states are not represented. Therefore, this mechanism is not applicable for coordination of active components.

We have implemented functional component coordination in OSGi by using BIP [6] coordination mechanisms. In BIP, systems are constructed by superposing three layers of modelling: *Behaviour, Interaction*, and *Priority*. The first layer, Behaviour, consists of a set of components modelled by FSM. The second layer models interaction between components defining explicitly which transitions can be synchronised. When several interactions are possible, priorities can be used as a filter. Interaction and Priority layers are collectively called *glue*. 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 all the components involved in it;
3. The notified components execute the functions associated to the corresponding transitions.

To use the transition synchronisation mechanism, developers must ensure that the component states remain stable during one cycle of the above protocol: *a component must be able to perform any transition it has announced as possible to the BIP Engine*.

As mentioned above, this approach allows a clear separation between the component behaviour and system-wide coordination policies. For coordination scenarios that require global state information (see for example our use case in Section 4.1), dedicated *monitor* components can be added in straightforward manner. This allows centralising all the information related to the coordination in one single location, instead of distributing it across the components. This 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.

We have taken a non-invasive approach relying on existing API for the interaction with the controlled components. With our approach, designers write a separate annotated Java class that we call *BIP Specification*. BIP specification is an abstract model of the component that is aware of its functional states. It defines the functions associated to the corresponding transitions. Transitions can be of three types: *enforceable*, *spontaneous* and *internal*. Enforceable transitions are used for coordination through the BIP Engine; spontaneous transitions are used to take into account the changes in the controlled component; finally, internal transitions can be used to make the BIP Specification more concise—when enabled, they are executed immediately. To ensure execution determinism, at most one internal transition can be enabled at any given execution step. BIP Specification developers are responsible for enforcing the validity of this requirement. However, our tools log an exception when a violation is detected at runtime.

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. The Behaviour object is used by the BIP Engine to coordinate the actions of the component with the other components of the system. Additional components can be provided in a similar manner to monitor and influence the system behaviour, in particular to impose safety properties. For specifying larger components, where annotations become impractical, we have defined a Behaviour API, which allows developers to construct a Behaviour object programmatically.

BIP coordination extension for OSGi has been implemented and tested in Connectivity Factory™, a flagship product of Crossing-Tech S.A. The main use-case consists in managing the memory usage by a set of Camel routes[1]. A Camel route connects a number of data sources to transfer data among them. The data can be fairly large and may require additional processing. Hence, Camel routes share and compete for memory. Without additional coordination, simultaneous execution of several Camel routes can lead to `OutOfMemory` exceptions, even when each route has been tested and sized appropriately on its own. We have designed an annotated BIP Specification for Camel routes, using the `suspend` and `resume` functions provided by the Camel Engine API. We also use notification policies provided by Camel to observe the spontaneous modifications of the route states. By introducing an additional Monitor component, we limit the number of Camel routes running simultaneously to ensure that the available memory is sufficient for the safe functioning of the entire system.

Using BIP allows taking into consideration the structure of the controlled software and also the coordination constraints imposed by the safety properties. The operational semantics of the BIP framework is implemented by a dedicated Engine used for the coordination of software modules according to the three-step protocol above. The BIP Engine is packaged as an OSGi bundle, using the mechanisms provided by OSGi to publish the service that can be used by the software modules.

This report is structured as follows. Section 2 explains the component model we used and how the proposed design methodology should be applied in practice. Section 3 describes the implemented software architecture. Section 4 illustrates the approach on two use cases. Section 5 discusses the related work.

2. Design Methodology

2.1. Component Model

We consider a system of components, each represented by a Finite State Machine (FSM) extended with ports. The FSM is specified by its states and guarded transitions between them. Each transition has a function and a port associated to it. One port can be associated to several transitions. There cannot be transitions from the same state labelled by the same port. 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;

[1] http://camel.apache.org/routes.html

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 component environment and can be performed immediately. Spontaneous transitions represent changes in the environment that affect the component behaviour, but cannot be controlled. Enforceable transitions represent the controllable behaviour of the component. At each state, at most one internal transition is allowed to be enabled.

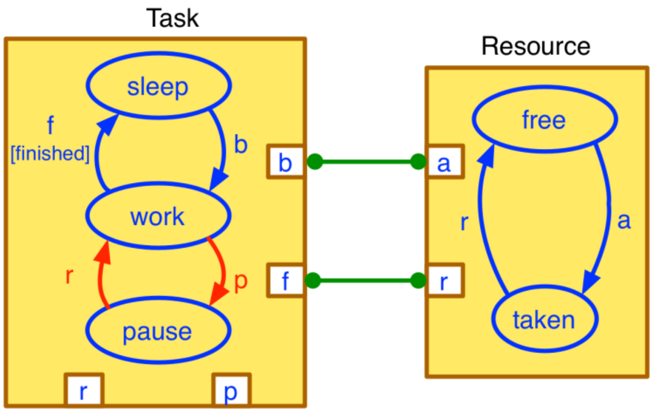


Figure 2: The model of the Task/Resource example

We illustrate the model on a simple example consisting of two components: a Task and a Resource (Figure 2). The Task component has states `sleep`, `work` and `pause`, enforceable transitions (blue arrows in Figure 2) labelled by ports `b` (begin) and `f` (finish) and spontaneous transitions (red arrows in Figure 2) labelled by ports `p` (preempt) and `r` (resume). The Resource component has states `free` and `taken` and corresponding enforceable transitions labelled by ports `a` (acquire) and `r` (release). In order to begin execution, the task has to acquire the resource. Upon termination of the task, the resource is released. This is achieved by imposing synchronisation of the corresponding enforceable transitions of the two components (see the green connectors in Figure 2). The Task can be spontaneously preempted and resumed due to changes in the environment (e.g. by a component with higher priority), which we do not model here. Finally, the Task can finish execution only when it has finished all the work it has been assigned. This is modelled by associating a boolean guard `finished` to the `f` (finish) transition of the Task component.

In our implementation, the execution of transitions is controlled and managed by a dedicated *BIP Executor* object, described in detail in Section 3.2. The synchronisation between components is ensured by the *BIP Engine*. BIP Executor maintains a queue of notifications corresponding to spontaneous transitions and cyclically executes the following two steps:

1. All transitions from the current state are checked for enabledness.
2. One transition is picked for execution (in order of decreasing priority):
   (a) If there is an internal transition, it is fired right away.
   (b) If there is a spontaneous transition, and the corresponding event has already happened (i.e. a corresponding notification is available in the queue), this spontaneous transition is performed. 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.

(c) If there are enforceable transitions, the Executor informs the Engine about the current state and the disabled ports (i.e. which transitions cannot be performed). The Executor then waits for a response from the Engine, indicating the port to execute. Upon receiving this response, the Executor 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 Executor will announce the enforceable transition to the Engine. Thus, in order to satisfy the state stability property mentioned in the Introduction, it must be possible to postpone the execution of the spontaneous transition until after the execution of the enforceable one. In other words, a spontaneous transition labelled with the same port must exist, leaving the target state of the enforceable transition.

2.2. Design steps

One of the benefits of our approach is that the developer does not need to access the existing source code or modify it. The design process involves two steps:

1. Defining the Behaviour for each component;
2. Specifying how the components can interact.

2.2.1. Specification of component behaviour

The FSM extended with ports, discussed in Section 2.1, is provided as an instance of a Java class implementing the `Behaviour` interface. This interface is used by the Executor and the Engine and it provides access to the information about states, ports, transitions and guards of the FSM. For storing the corresponding information, we have defined the `Port`, `Transition` and `Guard` classes (see below).

The following information must be provided:

- the name of the component;
- the list of all the states and the initial state;
- the list of ports (list of `Port` instances, where each instance is specified by its name and type—enforceable or spontaneous);
- the list of transitions (list of `Transition` instances, where each instance is specified by its name, source state, destination state and guard expression);
- the list of guards (list of `Guard` instances, where each instance is specified by its name and a method returning a boolean value, which corresponds to the actual guard function);
- a reference to the object, implementing the API used by the methods associated to guards and transitions of the component.

There are two different ways to provide this information. The first one consists in directly creating an object implementing the Behaviour interface. The second one is to use a mechanism provided for building such objects from source code *annotations*. Annotations are syntactic metadata associated to parameters, fields, methods or class declarations. We use annotations associated to class and method declarations. Several annotations can be associated to one declaration. They are processed by the Executor and the Behaviour object is created automatically.

We have defined the following annotations (cf. Figure 3):

- `@bipComponentType`—annotation associated to a BIP component specification class. It has two fields: `name`, the name of the component type, and `initial`, the name of the initial state.
- `@bipPort`—annotation associated to a BIP component specification class. It has two fields: `name`, port name, and `type`, port type which can be "spontaneous" or "enforceable", defining the type of associated transitions. The internal transitions are defined by omitting their name. We use an additional `@bipPorts` annotation to specify several instances of `@bipPort` annotation associated to the same class declaration.
- `@bipGuard`—annotation associated to a method used to compute a transition guard. The method should return a boolean value. This annotation has one field: `name`, guard name which can be referred to in a guard expression of a transition.
- `@bipTransition`—annotation associated to a transition handler method with four fields: `name`, the name of the port labelling the transition, `source`, the name of the source state, `target`, the name of the target state and `guard`, a boolean expression on the names of the guard methods (true if omitted). The guard expressions can be defined using parenthesis and three logical operators: negation (`!`), conjunction (`&`) and disjunction (`|`). Only ports defined by the `@bipPort` annotation can be used as transition names.

Figure 3 shows an annotated Java class defining the Task component from the example in Section 2.1.

```
@bipComponentType(initial = "sleep",
                     name = "org.bip.spec.Task")

@bipPorts({
      @bipPort(name = "b", type = "enforceable"),
      @bipPort(name = "f", type = "enforceable"),
      @bipPort(name = "r", type = "spontaneous"),
      @bipPort(name = "p", type = "spontaneous")
})

public class Task {
      //class declarations
      @bipTransition(name = "b", source = "sleep",
                      target = "work", guard = "")
      public doWork() {
            //method body
      }
      @bipTransition(name = "f", source = "work",
                      target = "sleep", guard = "finished")
      public goToSleep() {
            //method body
      }
      @bipTransition(name = "p", source = "work",
                      target = "pause", guard = "")
      public pauseWork() {
            //method body
      }
      @bipTransition(name = "r", source = "pause",
                      target = "work", guard = "")
      public resumeWork() {
            //method body
      }
      @bipGuard(name = "finished")
      public boolean operationFinished() {
            return operationFinished;
      }
}
```

Figure 3: Annotations for the Task/Resource example (cf. Figure 2)

2.2.2. Specification of interaction constraints

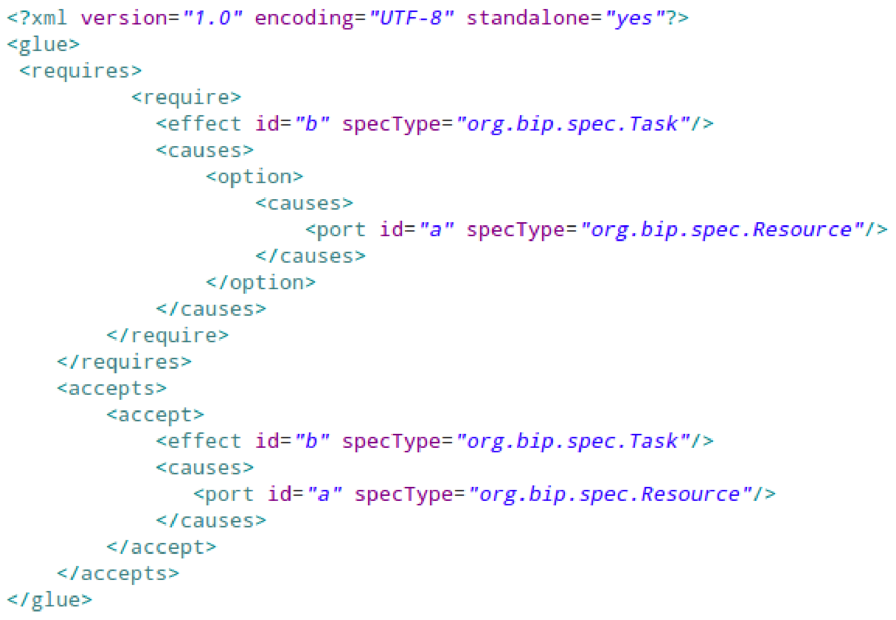
To define the interaction model, the developer specifies the interaction constraints of each component. An interaction constraint can be provided for each port of a system. Two types of constraints can be used to define allowed interactions:

- **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 `Task.b Require Resource.a` forces the port `b` of any component of type `Task` to synchronise with a port `a` of some component of type `Resource`. Furthermore, the constraint `Task.b Accept Resource.a` 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. Figure 4 shows interaction constraints for the Task component of Figure 2.



```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<glue>
 <requires>
         <require>
           <effect id="b" specType="org.bip.spec.Task"/>
           <causes>
               <option>
                   <causes>
                       <port id="a" specType="org.bip.spec.Resource"/>
                   </causes>
               </option>
           </causes>
        </require>
    </requires>
    <accepts>
        <accept>
            <effect id="b" specType="org.bip.spec.Task"/>
            <causes>
               <port id="a" specType="org.bip.spec.Resource"/>
            </causes>
        </accept>
    </accepts>
</glue>
```

Figure 4: Interaction constraints for the Task/Resource example (cf. Figure 2)

2.3. Run-time application setup

Components must be aware of the BIP Engine they are working with and the BIP Engine must be informed of the components it coordinates. For that purpose, a number of methods are defined in the `BIP Engine` and `BIP Executor` interfaces. The registration process can be performed as follows:

- Register each component so that all Executor instances are aware of the BIP Engine. This is done by calling the `register` method of the `BIP Executor` interface:

  `executor.register(engine);`

  This method, in turn, calls the `register` method of the BIP Engine so that the Engine gets informed of each component it coordinates:

  `engine.register(this);`

- Provide to the Engine the interaction specifications by calling the method `specifyGlue` of the `BIP Engine` interface. The argument of the method is an instance of a dedicated Java object which contains the system interaction constraints as defined in Section 2.2.2:

```
engine.specifyGlue(bipGlue);
```

- Start the BIP Engine coordination mechanism by calling the `execute` method of the `BIP Engine` interface:

```
engine.execute();
```

3. Implementation

3.1. Architecture

The software architecture of the proposed framework is shown in Figure 5. The architecture consists of two major parts: the part that involves the components to control (on the left) and the BIP Engine part (on the right). The grey boxes in the diagram represent OSGi bundles.

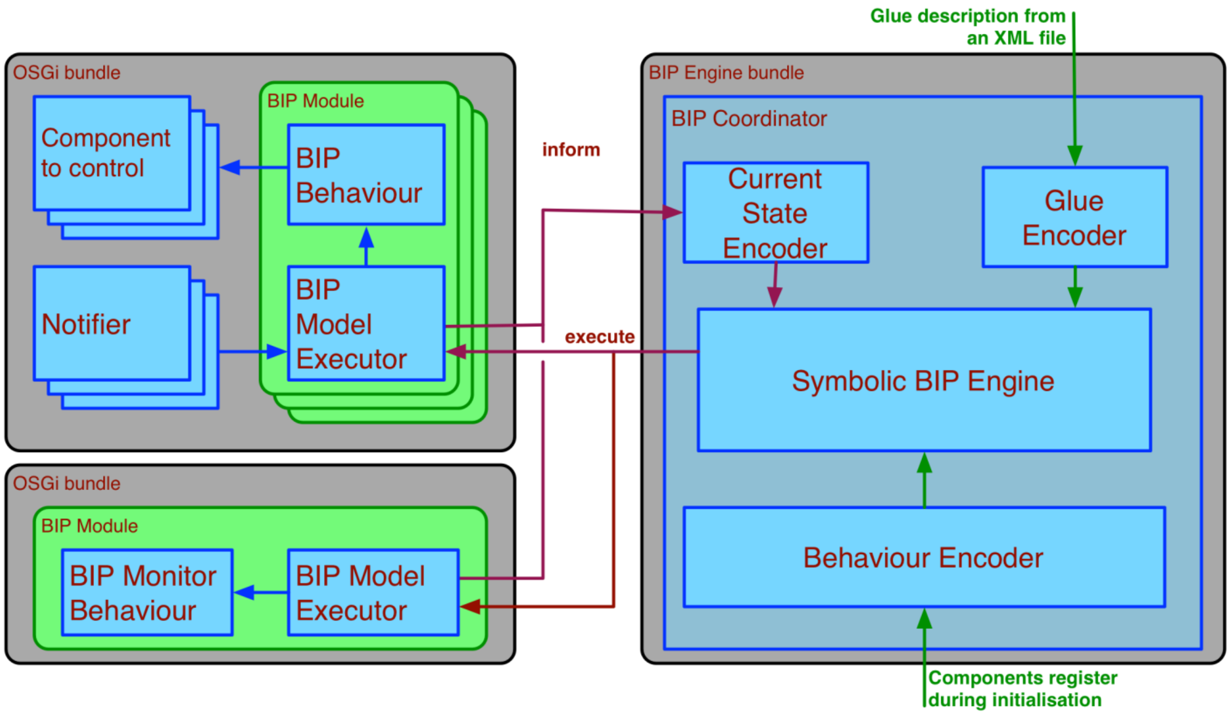


Figure 5: Software architecture

For each component a BIP Behaviour is generated at runtime. The Behaviour contains all the information about the FSM and ports of the controlled entities. As shown in the left-hand part of the diagram, 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 informing the BIP Executor of spontaneous events relevant to the controlled entities.

The right-hand part of Figure 5 presents the BIP Engine, which coordinates the execution of the components. The implementation of the Engine is modular and makes use of Binary Decision Diagrams (BDDs) [3].[2] BDDs are efficient data structures to store and manipulate boolean formulas.

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

The ports and states of all components of the system are associated to boolean variables. These boolean variables are used at initialization by the *Behaviour* and *Glue Encoders* to translate the corresponding constraints into boolean expressions (green arrows in Figure 5). At every execution cycle during runtime, each component provides the information about its current state and enabled ports (the red inform arrow). This information is also encoded into boolean formulas by the *Current State Encoder*. The Symbolic BIP Engine computes the conjunction of these constraints to obtain the global boolean formula that represents the possible interactions of the system.

In the following sections, the implementations of the BIP Executor and BIP Engine are presented in more detail.

3.2. BIP Executor

The BIP Executor builds the Behaviour object from the component specification or receives it as an argument at the initialisation phase (as specified in Section 2.2.1). At each execution cycle, the Executor interprets the Behaviour and fires the transitions. In order to fire a transition, the Executor performs three steps:

1. Using the Behaviour, retrieves the method corresponding to the transition;
2. Using Java Reflection mechanism [13], invokes this method;
3. Updates the current state of the Behaviour to the target state of the performed transition.

There can be at most one internal transition enabled at the current state. If no internal transitions are enabled, the choice of spontaneous or enforceable ones is made based on the information obtained from the notifiers and the Engine respectively (cf. Section 2.1):

- **Spontaneous:** a Notifier entity sends a notification (i.e. the name of a spontaneous port) to the Executor via the function `inform`; this port is then stored in a queue waiting to be processed;
- **Enforceable:** the Engine sends the name of an enforceable port to the Executor via the function `execute` (the red arrow in Figure 5), the execution is performed immediately.

BIP Executor implements the protocol presented in Section 2.1. In order to ensure consistency of guard valuations, the guard functions are computed only once for each execution cycle, regardless of how many times they are used.

3.3. BIP Engine

The BIP Engine orchestrates the coordination of the components by deciding which interaction should be fired given the information about the current states of the components. To do that, the Engine applies the three-step protocol presented in the Introduction in a cyclic manner.

The Engine is packaged as an OSGi bundle and provides the coordination service used by BIP Executor components. Furthermore, its implementation is modular and consists of five main parts (cf. Figure 5): three Encoders, the BIP Coordinator and the core Symbolic BIP Engine.

The BIP Coordinator manages the flow of information between the components and the Symbolic BIP Engine through a dedicated `BIP Engine` interface. Part of this interface is used by the BIP Executors, whereas the other part is used by the Symbolic BIP Engine. In particular, the BIP Coordinator receives:

- The Behaviour during the registration of the components at initialisation phase and sends it to the Behaviour Encoder (`register` method presented in Section 2.3).
- The interaction constraint specifications, which are external to the components and are provided only at initialization (`specifyGlue` method presented in Section 2.3). The specifications are provided as an instance of a special `bipGlue` object and the BIP Coordinator forwards them to the Glue Encoder.
- For each component, the current state and the list of ports disabled by guards. This information is provided at each execution cycle (red `inform` arrow in Figure 5) and is forwarded to the Current State Encoder.
- The chosen interaction from the Symbolic BIP Engine. The BIP Coordinator instructs the components to make the necessary transitions (red `execute` arrow in Figure 5).

The three encoders compute the boolean representation of the information they receive from the BIP Coordinator. The boolean representation of the behaviour and interaction constraints is only computed once at initialisation, whereas the boolean representation of the current state information is recomputed at each execution cycle. The Symbolic BIP Engine receives all the information provided by the three encoders. At each execution cycle, it computes the set of feasible interactions in the system, chooses one interaction among these and notifies the BIP Coordinator.

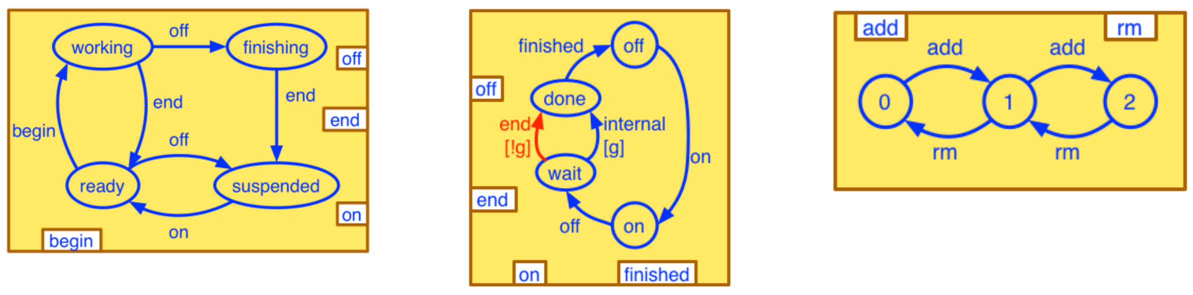
3.4. Integration with OSGi

In order to integrate our framework within an OSGi container the following bundles were created:

- BIP API bundle, providing the interfaces to be used within the framework;
- BIP Engine bundle, providing the Engine services;
- BIP Executor bundle, providing the Executor code;
- BIP Admin bundle, providing the commands for setting up the application in the Connectivity Factory™ OSGi container.

The separation into bundles allows us to isolate the aspects of our framework and improves modularity. BIP Admin bundle does the binding between the components and the BIP Engine, deserialises the XML representation of interaction constraints and allows the user to manage the execution by providing the commands `bip:register`, `bip:execute` and `bip:stop` for the application setup presented in Section 2.3. To process the commands this bundle uses Karaf [3]—a generic platform providing higher level features and services specifically designed for creating OSGi-based servers.

[3] http://karaf.apache.org/



(a) Camel Route **(b) Switchable Route** **(c) Monitor**

Figure 6: The models of the Camel Route, Switchable Route and the Monitor

4. Use-case scenarios

In this section, we describe two examples used to validate our approach. The Switchable Routes use-case is a simplified real-life application encountered by Crossing-Tech S.A. We use the Towers of Hanoi example to illustrate a slightly more complex coordination scenario.

4.1. Use-case 1: Switchable Routes

This use-case consists in managing the memory usage by a set of Camel routes. A Camel route connects a number of data sources to transfer data among them. The data can be fairly large and may require additional processing. Hence, Camel routes share and compete for memory. Without additional coordination, simultaneous execution of several Camel 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.

In the example below, we consider three routes connected in a cycle. We seek to ensure that at most two routes are active simultaneously.

An abstract model of the route is presented in Figure 6(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 the processing of a file. The route can be turned off via `off` transition from states `ready` and `working`.

Notice that this model does not respect the state stability assumption: spontaneous transition `begin` from the state `ready` cannot be postponed if the notification arrives after the component has promised to perform the `off` transition. To address this problem, we define another model shown in Figure 6(b). This model is obtained by merging the `ready` and `working` states together (the `on` state) and splitting the `finishing` state in two: `wait` and `done`.

Firing the enforceable transition `off` takes the route into the 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`. Since the transition `finished` from the state `done` is enforceable, it requires a communication with the Engine to be fired. This is used to coordinate the enforceable transitions of other components.

Finally, we introduce an additional Monitor component, shown in Figure 6(c), which allows us to limit the number of simultaneously active routes to two. This is achieved by enforcing, for each route component, the synchronisation between its port `on` (respectively `finished`) and the port `add` (respectively `rm`) of the Monitor. We use interaction constraints similar to the ones of the Task/Resource example of Section 2.1.

Figure 7 shows the execution log of the system consisting of three Switchable Routes and one Monitor with component IDs 393–396 respectively. These IDs are assigned by the OSGi registry. The components use the `inform` function described in Section 3.3 to provide to the Engine the information about their respective current states. In particular, in Figure 7, two out of the three Switchable Routes inform the BIP Engine that they are at state `done` and therefore the Monitor correctly informs the Engine that it is in the state corresponding to the number of active Switchable Routes.

In the next step, the Engine selects the interaction `finished•rm`, which forces the Monitor to decrement the counter due to the completion of the ID 395 route.

```
***************************** Inform ***********************
Component: switchableRoute395 informs that is at state: done
Component: switchableRoute393 informs that is at state: off
Component: switchableRoute394 informs that is at state: done
Component: monitor396 informs that is at state: 2
************************************************************
*************************** Engine *************************
ChosenInteraction:
Chosen Component: switchableRoute395
Chosen Port: finished
Chosen Component: monitor396
Chosen Port: rm
************************************************************
***************************** Inform ***********************
Component: switchableRoute395 informs that is at state: off
Component: switchableRoute393 informs that is at state: off
Component: switchableRoute394 informs that is at state: done
Component: monitor396 informs that is at state: 1
************************************************************
```

Figure 7: BIP Engine: One execution cycle printout of the Switchable routes

4.2. Use-case 2: The Towers of Hanoi

In this use-case, the system consists of three pegs: the left, the middle and the right. The left peg holds disks of size decreasing from *n* at the bottom to 1 at the top. The objective is to transfer all disks to one of the other pegs, moving only one disk at a time and never putting a larger disk on top of a smaller one. We refer to the move, where we put a disk on an empty peg or on top of a larger disk, as a *legal move*.

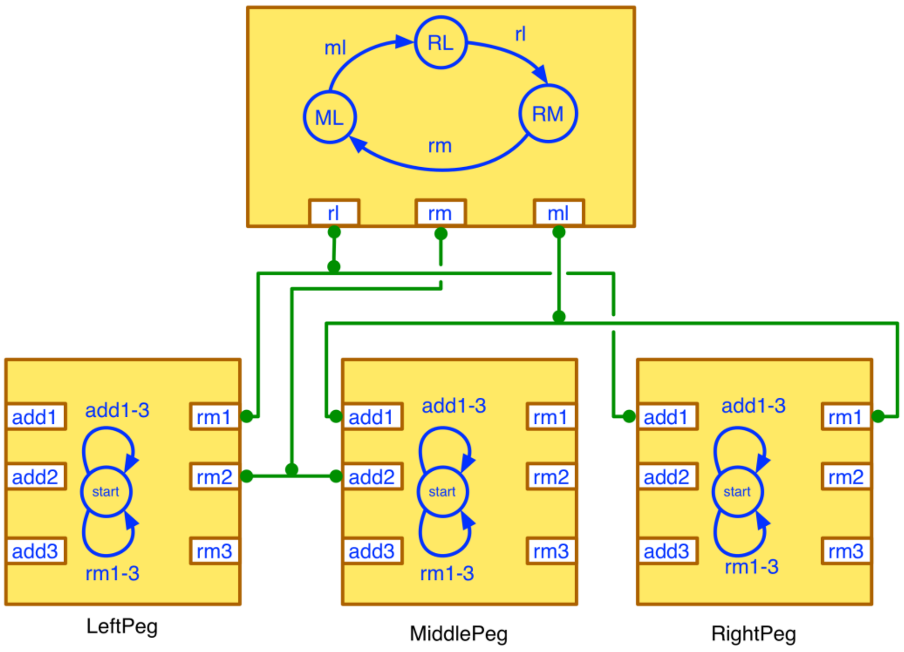


Figure 8: The Towers of Hanoi use case

The algorithm is different for odd and even numbers of disks. A total of $2^n - 1$ moves are made in each case. Here we only present the model that solves the Towers of Hanoi for odd values of *n*. The model for even values is similar.

The system model is shown in Figure 8. It has one component for each of the three pegs: left, middle and right. In addition, there is a Monitor component ensuring that only legal moves are made between pegs. For the sake of clarity, we do not show all the connectors in Figure 8. More precisely, we only show the connectors that correspond to the interactions of the system for the first three movements of the algorithm, when all the disks are on the left peg.

Behaviours of all peg components are identical. They differ only at initialisation: the left peg is initialised to be full, the other two to be empty. Each peg has 2*n* ports: $\texttt{add}_1$, …, $\texttt{add}_n$ for adding the corresponding disk and $\texttt{rm}_1$,…,$\texttt{rm}_n$ for removing. Each peg has only one state `start` and a local boolean array (not shown in the figure) to keep track of the disks it holds. Each transition has the corresponding guard associated to it (also not shown in the figure), which determines whether the corresponding disk can be added to (respectively removed from) the peg.

The behaviour of the Monitor component, combined with the interaction constraints imposed by the connectors, ensures that a disk is moved between the left and the middle pegs (transition `rm`), then between the middle and the right pegs (transition `ml`), then between the left and the right pegs (transition `rl`) and so on.

Partial trace of the system execution is shown in Figure 9. As expected, the application correctly terminates after $2^n - 1$ moves.

```
************************** Inform ***************************
Component: org.bip.spec.MiddleHanoiPeg
informs that is at state: start
disabled ports: piece1Remove, piece2Remove, piece3Remove
**************************************************************
************************** Inform ***************************
Component: org.bip.spec.HanoiMonitor
informs that is at state: state-RM
*************************************************************
************************** Inform **************************
Component: org.bip.spec.RightHanoiPeg
informs that is at state: start
disabled ports: piece1Add, piece2Add, piece2Remove, piece3Add
**************************************************************
************************** Inform ***************************
Component: org.bip.spec.LeftHanoiPeg
informs that is at state: start
disabled ports: piece1Remove, piece2Add, piece3Add, piece3Remove
**************************************************************
************************** Engine ***************************
ChosenInteraction:
Chosen Component: org.bip.spec.MiddleHanoiPeg
Chosen Port: piece2Add
Chosen Component: org.bip.spec.LeftHanoiPeg
Chosen Port: piece2Remove
Chosen Component: org.bip.spec.HanoiMonitor
Chosen Port: rm
**************************************************************
************************** Inform ***************************
Component: org.bip.spec.LeftHanoiPeg
informs that is at state: start
disabled ports: piece1Remove, piece2Remove, piece3Add
**************************************************************
************************** Inform ***************************
Component: org.bip.spec.RightHanoiPeg
informs that is at state: start
disabled ports: piece1Add, piece2Add, piece2Remove, piece3Add
**************************************************************
************************** Inform ***************************
Component: org.bip.spec.MiddleHanoiPeg
informs that is at state: start
disabled ports: piece1Remove, piece2Add, piece3Add, piece3Remove
*************************************************************
**************************Inform ***************************
Component: org.bip.spec.HanoiMonitor
informs that is at state: state-ML
*************************************************************
```

Figure 9: BIP Engine: One execution cycle printout of the Hanoi towers

5. Related work

Different approaches have been proposed to deal with the coordination of concurrent systems. First of all, locks and semaphores [15] 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 was developed by the OSGi community [18] allowing simple coordination between multiple software components. A Coordinator object has only two states: Active and Terminated. This approach provides developers with a higher abstraction level primitive for multi-party synchronisation barriers. Thus, some simple coordination can be ensured on several entities having no information about each other.

A different approach is taken by the AKKA library [14], which is based on the primitives of the Actor model [2]. Actors are concurrent components that communicate through ports. By relying on asynchronous communication, the actor model also avoids the use of low-level primitives, such as locks and semaphores. However, component coordination through the specification of 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 [12] 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 [12] framework also adopts an actor-oriented view of a system. Ptolemy actors can be hierarchically interconnected and support heterogeneous, concurrent modelling and design. However, for both of these frameworks, we are not aware of complementary work on using these 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 semantics proposed 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 behavioural 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 point-cuts 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. [16]. The authors propose specifying OSGi services based on a combined use of interface automata [11] and process algebra [5]. Their specification of services is centred on finding the best candidates for service substitution. Qin et al. [19] 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.

6. Conclusion

The introduction of OSGi was a tremendous improvement for the design of modular Java-based systems. OSGi greatly simplifies the work of a software developer and its benefit has been shown by the ever growing community of users. It takes into account such aspects as class loading, class visibility, bundle lifecycle and service dynamicity. However, the lifecycle layer makes a very strong simplification by limiting all the information about an active bundle to a single state. In Crossing-Tech's experience with OSGi, this happens to be very restrictive. In our practice, common coordination issues are very difficult to address with the mechanisms provided by the OSGi lifecycle layer. Developers had to resort to ad-hoc solutions to ensure that resources such as memory within the JVM running an OSGi container are not being exhausted.

The specification of what are the allowed global states and global state transitions is an integral part of the specification of a modular system. Using BIP, we have shown how these aspects can be taken into account in a non-invasive manner and without any impact on the technology stack within an OSGi container following the best practices of OSGi.

Although differentiating multiple components within a bundle could go beyond the desired scope of the OSGi specification, one has to notice that this finer granularity is already quite common, since OSGi best practices prefer packages to bundles as means to express dependency relationship. 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.

In this report, we have presented our approach to adding BIP coordination to OSGi. We have presented 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 formulas and three encoders that translate component and glue specifications into such formulas. We have presented two use cases illustrating our approach.

On-going 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. We also plan to define a more flexible and user-friendly format for specifying interaction constraints.

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 Berlin Heidelberg, 1997.

[2] G. Agha. Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge, 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. Theoretical Computer Science, 335(2-3):131–146, May 2005.

[6] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis. 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. Schatz. Behavioral specification based runtime monitors for OSGi services. In Proceedings of the 5th international conference on Leveraging Applications of Formal Methods, Verification and Validation: technologies for mastering change - Volume Part I, ISoLA'12, 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 and J. Sifakis. The algebra of connectors—structuring interaction in BIP. IEEE Transactions on Computers, 57(10):1315–1330, 2008.

[10] 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.

[11] 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.

[12] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity: The Ptolemy approach. Proceedings of the IEEE, 91(1):127–144, 2003.

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

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

[15] D. Lea. Concurrent Programming in Java: Design Principles and Patterns. Addison-Wesley, Reading, MA, 2 edition, 1999.

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

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

[18] OSGi Alliance. Coordinator service, http://www.osgi.org/javadoc/r5/enterprise/org/osgi/service/coordinator/Coordinator.htm l. (Accessed on 22/10/2013.).

[19] Y. Qin, H. Hao, L. Jim, G. Jidong, and L. Jian. An approach to ensure service behavior consistency in OSGi. In Software Engineering Conference, 2005. APSEC'05. 12th Asia-Pacific, 2005.